并行图论算法

SIMD

MIMD

VLSI



并行图论算法

唐策善　梁维发　　编著

国家八六三计划资助项目

中国科学技术大学出版社

1991・合肥

内 容 简 介

本书反映了并行处理应用于图论领域研究的最新进展。主要内容包括：并行计算模型；图的搜索、求连通分支、最小生成树、最短路径等一些基本图论问题的并行算法和分布式算法；求基本回路、双连通分支、关节点和桥、欧拉路径和哈密顿路径，AOE网和最大流，极大独立集和图着色等问题的并行和分布式算法；并行矩阵乘法在图论中的应用；组合搜索的并行算法；以及神经网络在图论中的应用。

读者对象：计算机及其应用专业、应用数学专业研究生和高年级学生，以及从事计算机、应用数学、网络工程、最优化设计等领域的科学研究和工程设计的科技工作者。

并行图论算法
康策善　梁维发　编著
*
中国科学技术大学出版社出版
(安徽省合肥市金寨路96号，邮政编码：230026)
中国科学技术大学印刷厂印刷
安徽省新华书店发行
*
开本：787×1092/16　印张：　20.25　字数：　480　千
1991年10月第1版　1991年10月第1次印刷
印数：1—3000　册
ISBN7-312-00316-8/TP・38
[皖]第08号　定价：11.0元

前　　言

通常，图是由顶点集合和边集合组成的，用它表示的问题就是所谓的图论问题．它广泛地出现在计算机科学、信息科学、人工智能、网络理论、系统工程、运筹学、控制论、物理学、化学、心理学、语言学、经济管理等领域，因而研究图论问题及其解法具有极其重要的理论和实际意义。

很多图论问题表达容易，但求解却很困难，常常需要花费大量的计算时间和存贮空间．现已证明相当多的图论问题是 NP-完全的，即当问题的规模足够大时，往往出现"指数爆炸"现象，这是传统的顺序计算机无法求解的。随着 VLSI 技术的发展和并行计算机的出现，为人们快速求解图论问题提供了一条新的途径．运用多处理机系统来求解图论问题，在科学上和工程应用中发挥了重要的作用。

形式地讲，并行算法是一些可同时执行的诸进程的集合，这些进程相互作用和协同动作，从而达到对给定问题的求解．读者将会看到：并行算法和计算机结构密切相关，不同结构的并行计算机将会导致不同风格的并行算法．但是，本书不可能研究所有具体的并行计算机上的并行图论算法，只能从现有的并行计算机中抽象出若干典型的并行计算模型，并围绕这些模型讨论各种图论问题的并行算法。

本书描述在多处理机系统上协同求解图论问题的并行和分布式算法，这些算法并不完全是现行的顺序图论算法简单的推广和扩充。事实上，有的图论问题可以从顺序算法转换为相应计算模型上的有效并行算法；有的图论问题则不然，必须从问题本身出发，结合并行计算模型，研究和设计新的并行算法．在各种计算模型上，如何设计和分析并行图论算法，是本书写作的宗旨，我们将通过众多图论问题并行算法的讨论，向读者介绍若干基本的研究方法和技术。

最近十余年来，并行处理应用于图论领域的研究异常活跃，每年都涌现出大量的学术论文和研究报告．在这浩如烟海的文献中，摄取哪些材料作为本书的基本内容，才能反映这一领域的最新成果和发展趋势，是我们的努力方向．作者通过对计算机系研究生、高年级大学生和青年教师讲授本书部分内容的教学实践，以及从事这一领域研究工作的切身体会，编著成书，奉献给广大读者．

本书侧重于非数值计算方面的内容。全书共分十四章。前两章是并行算法的基础，包括并行计算机与并行计算模型；并行算法的度量与设计技术。第三章到第六章集中介绍基本图论问题的并行和分布式算法。它们是：图的搜索；求图的连通分支；计算加权图的最小生成树；以及找图的最短路径等。第七章讨论并行矩阵乘法及其在图论算法中的应用。第八章到第十二章，叙述了图论的基本性质和一些较难的图论问题的并行和分布式算法。内容包括：无向图的基本回路、双连通分支、关节点和桥；欧拉路径和哈密顿回路；AOE 网和最大流；极大独立集和图的着色等问题的算法。第十三章讨论了用状态空间树表示的组合搜索问题的并行算法；最后一章研究了神经网络在图论中的应用。每章的结尾还给出了小结和参考文献，简要概述了该章的主要内容，列出了有关的最新进展，以便读

者进一步研究时参考.

本书前六章和第十三章由唐策善提供初稿，第七章至第十二章由梁维发提供初稿，第十四章由唐锡南提供初稿。最后，经唐策善统一修改后定稿。

本书是并行算法丛书之一，受国家 863 计划项目资助。

我们在撰写中，曾直接或间接引用了许多专家、学者的文献；陈国良教授对本书的编写和出版非常关心，仔细审阅了全书内容，提出过许多宝贵意见；唐锡南付出了辛勤的劳动，除了提供第十四章初稿外，对其它章节亦提出了不少有益的建议；唐锡晋帮助整理了本书的部分章节；陈一栋、贾南在毕业实习过程中为本书第十三章做了部分工作；本校教务处激光照排中心的同志们为本书的计算机录入和排版做了大量的工作，作者谨此一并致以诚挚的谢意。

当国内外系统、深入地阐述这方面的教材或专著尚属鲜见的时候，我们为能向读者奉献此书而感到高兴；同时，也深感水平所限，书中缺点和错误在所难免，恳请广大读者批评指正.

作　者

1991 年 5 月于中国科学技术大学

目　录

第一章　并行计算机与并行计算模型

1.1 并行计算机及其分类

计算机发展的趋势是运算速度越来越快，存贮容量越来越大，软件越来越丰富，体系结构越来越完善，因此，处理能力越来越强．从早期的简单数据处理到近年来的复杂知识处理，都表明了这一点．计算机发展的历史表明，为了达到更高的处理性能，除了提高元器件的速度外，系统结构也必须不断改进，特别是当元器件速度达到极限（比如光速）时，后者将成为问题的焦点．计算机系统结构的改进，主要是围绕在同一时间间隔内增加操作量，即所谓的**并行处理**(Parallel Processing)技术．为了并行处理而设计的计算机系统称为**并行计算机**(Parallel Computer)．在并行计算机上求解问题的过程称为**并行计算**(Parallel Computing)．在并行计算机上设计求解给定问题的算法称之为**并行算法**(Parallel Algorithm)．

并行计算是一个比较年轻的领域：因为著名的阵列处理机 Illiac Ⅳ 1975 年才开始运行；第一台向量流水机 Cray−1 1976 年才交付使用；尤其是 VLSI 的发展，使得计算机的价格急剧下降，由多个处理器组成的并行计算机才流行起来．

并行计算机的发展主要是某些大型应用领域的要求：如气象预报，空气动力学，人工智能，卫星图象处理，核反应堆数据处理以及军事上的应用等．为了达到上述高性能要求，除了提高线路及器件速度外，主要是改进计算机的系统结构，例如：

(1) **引入 I／O 通道**：将费时的 I／O 操作交给 I／O 处理器（通道）去做，从而使 CPU 集中在计算上；

(2) **交叉存贮**：将存贮体分成多个模块，以达到并行存取和减少存取冲突为目的；

(3) **高速缓冲存贮器**：减少主存与处理器的数据交换时间，平滑这两者之间的数据流动速率；

(4) **指令先行**：一次取出适当多条将要执行的指令，使得取指令可以与指令译码同样快；

(5) **多功能单元**：在中央处理器中设置多个功能单元（加法、乘法器等）可以提高单一程序的吞吐量，缩短周转时间；

(6) **指令重叠**和**流水线技术**：在同一时间允许多条指令在不同的阶段按流水方式执行不同的操作；

(7) **向量处理技术**：一条向量指令可以同时处理 n 个分量，它们可以同时在各个处理器中进行运算，也可以连续地送入流水线中进行重叠处理；

(8) **多道程序设计**：同时把若干作业放在内存中，允许在同一时间有多道程序处于运行状态；

(9) **分时系统**：允许多个终端用户同时交互地使用同一台计算机；

(10) **数据驱动**：与冯・诺依曼计算机不同，它不是采用指令驱动操作，而是采用操作数（据）驱动操作，这样如有多个操作数准备就绪时，就可以彼此并行地执行而不受指令顺序执行的限制，从而可以充分开拓并行度．

1.1.1. 并行计算机介绍

首先我们介绍现有的一些典型并行计算机，以增加大家的感性认识。这里不去描述细节，仅仅讨论一下并行计算机的类型、结构特点和运行方式。

1. 阵列处理机(Array Processor)[5]

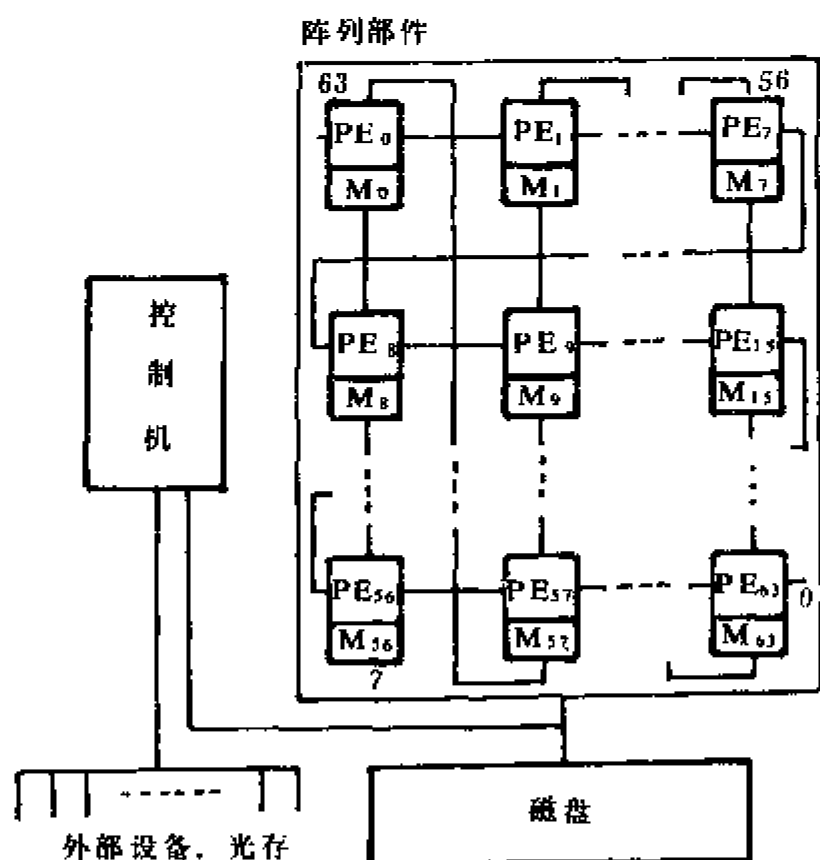


图 1.1 Illiac IV 机器框图

有名的 Illiac IV 就是阵列结构的。如图 1.1 所示，64 个 PE(Processing Element)各自带有局部存贮器 M，它们排成 8×8 的阵列。每个 PE 均可和它的上、下、左、右四个相邻的 PE 相连。所有 PE 在同一控制器控制下，按同一指令要求对不同的数据进行操作，从而达到操作级并行。

2. 流水线处理机(Pipeline Processor)

将生产流水线装配技术应用于计算机结构中，把计算机的运算部件或控制部件等装配成一些有序的子部件，利用功能部件分离与时间重叠办法，使每个被操作对象处在整个操作流程的不同功能部件中，且保持在不同的完成阶段，从而达到操作级的并行，这种结构的计算机称之为**流水线处理机**，亦叫**向量机**。流水线思想首先应用于指令的操作上；继之在功能划分的基础上，以流水线方式组成高速中央处理器；进而出现了在一台机器的中央处理器设置多条专用流水线，让它们并行工作或协同完成一些复合操作。这种系统对向量加工甚为有效，是目前解决大型数值计算问题巨型机的主要型式。

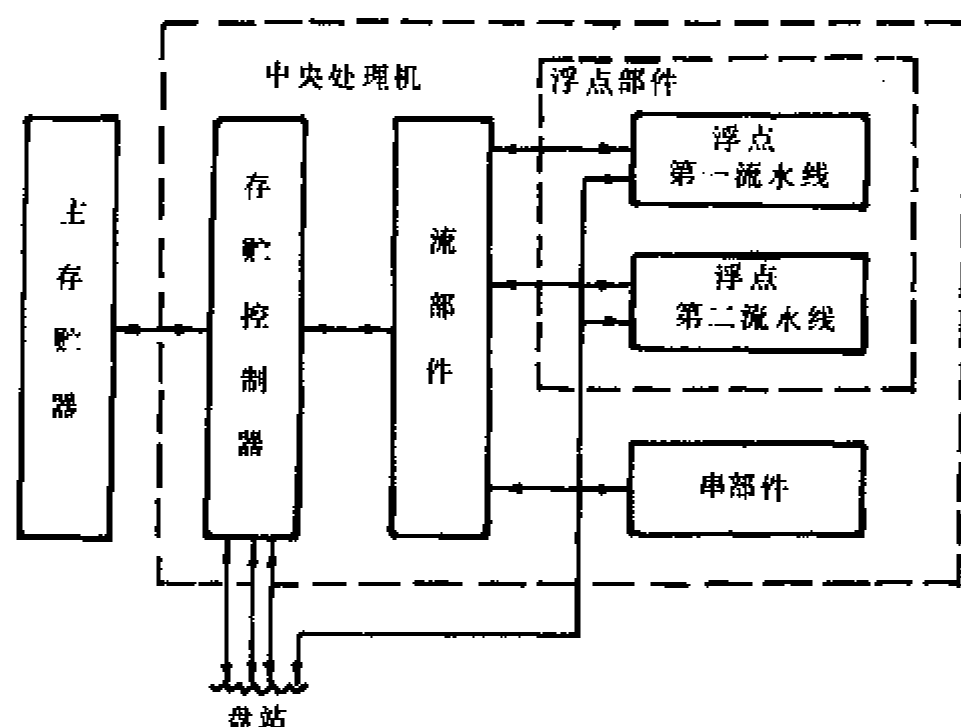


图 1.2 Star-100 机器框图

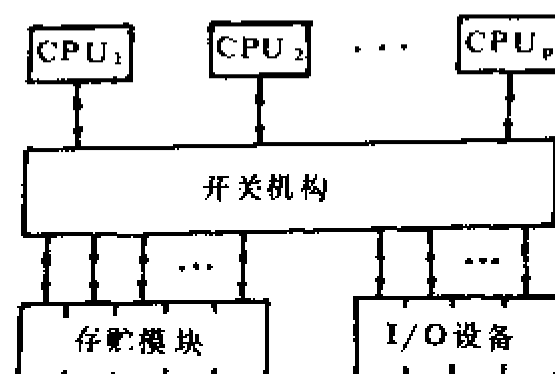


图 1.3 紧耦合的多处理机系统

Star−100 是典型的流水线向量处理机[6]．如图 1.2 所示，它由 CPU、主存、外围机及盘、站和外部设备组成．

3. 多处理机(Multiprocessor)和多计算机(Multicomputer)

多处理机（多处理器）和多计算机是由一些可编程的且可各自执行自己程序的多个处理器组成．其中，多处理机以各处理器共享公共存贮器为特征；而多计算机以各处理器经通信链路传递消息为特征．它们与阵列机的根本区别在于：阵列机中每个处理器只能执行中央处理器的指令，而后者可以执行自己的指令，这样可以达到指令级、任务级的并行．

如图 1.3 所示，在多处理机系统中，如果所有处理器都通过一个中央开关机构（如公共总线、交叉开关，包开关等）去访问全局共享存贮器，则这种形式的多处理机称之为**紧耦合多处理机**(Tightly Coupled Multiprocessor)系统，美国卡内基−梅隆大学的 $C._{mmp}$ 就是一个著名的紧耦合多处理机系统[7]。同紧耦合多机系统不同，若每个处理器各自有一个局部存贮器，所有的局部存贮器地址空间加在一起形成整个共享存贮空间，则这样的多处理机系统称为**松散耦合多处理机**(Loosely Coupled Multiprocessor)系统。因为松散耦合的多处理机没有集中的开关机构，所以可连接大量的处理器．卡内基一梅隆大学的 C_m^* 是另一个著名的松散耦合的多处理机系统[8]。如图 1.4 所示。

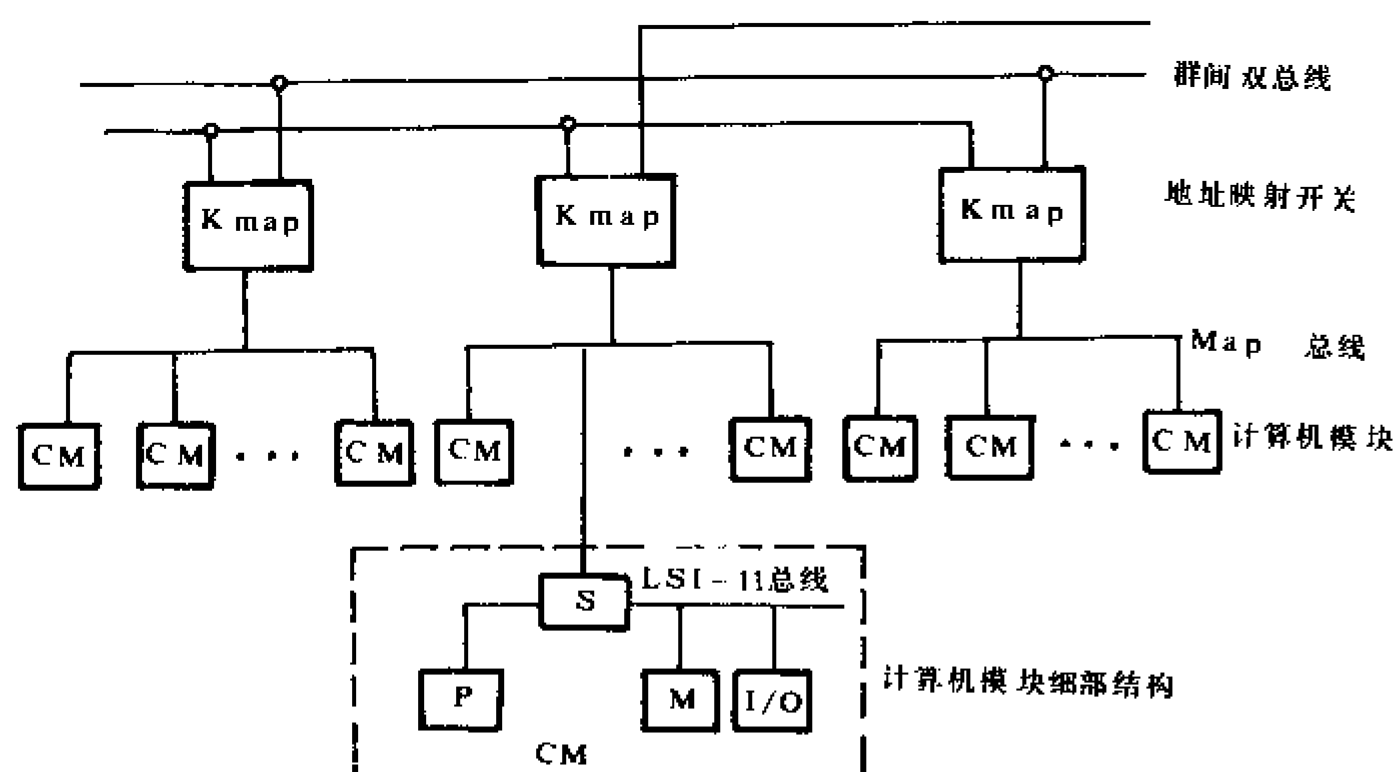


图 1.4 C_m^* 多处理机结构框图

C_m^* 采用了机群(Group)结构．目前已研制成的系统共有 50 台 DEC LSI−11 微型计算机，用三级总线(LSI−11 总线、Map 总线以及群间双总线)连接起来，其中 K_{map} 负责把各个计算模块所带有容量为 28K 字的局存组织在一起，形成统一的具有 28 位地址的虚拟存贮空间．

上面我们介绍的几种并行计算机基本上都还是在冯・诺依曼定义的框架内．目前已发展了许多非冯・诺依曼机，如**数据流计算机**(Data Flow Machine)，**归约计算机**(Reduction Machine)，**推理计算机**(Inference Machine)以及**神经网计算机**(Neural Networks Machine)等，虽然都是新一代计算机，但目前在这些机器上研究的并行算法尚未成熟，除了神经网机外，其它的不在介绍之列．

1.1.2 并行计算机分类

1. Flynn 分类法[9]

1966 年，M.J.Flynn 提出了著名的 Flynn 分类法。他按指令流及数据流将计算机系统分为四大类。

(1) **单指令流单数据流**：SISD(Single Instruction Stream Single Data Stream)计算机，这就是传统的串行计算机；

(2) **单指令流多数据流**：SIMD(Single Instruction Stream Multiple Data Stream)计算机，上一节所述的阵列机，流水线处理机均属于此类计算机。

(3) **多指令流单数据流**：MISD(Multiple Instruction Stream Single Data Stream)计算机。此类计算机是否存在尚有疑议。

(4) **多指令流多数据流**：MIMD(Multiple Instruction Stream Multiple Data Stream)计算机，上一节所述的多处理机和多计算机均属此类计算机。

2. Handler 分类法[10]

1977 年，Handler 根据计算机系统中流水线和并行度出现的级别，将一台计算机表示为三对整数。令 PCU 代表处理器控制单元，它相当于一个处理器或 CPU；ALU 代表算术逻辑运算单元，它相当于功能单元或处理单元；BLC 代表位一级电路。于是，按照 Handler 分类法，一台计算机可表示为：

$$T(C) = \langle K\times K', D\times D', W\times W'\rangle$$

其中：K = PCU的数目；

K' = 能够流水执行的PCU数目；

D = 每个PCU所控制的ALU数目；

D' = 能够流水执行的ALU数目；

W = ALU或处理单元PE中的位数；

W' = 在所有ALU或单个PE中流水线段数。

上式中，如果任一对整数的第二个元素值为 1，则就略去它。在单一计算机系统中，符号"×"可用于描述不同种类处理器的连接。

如 CDC 6600 计算机，它有一个 CPU；ALU 有 10 个功能单元，它们都可以流水地执行；CDC 6600 字长 60 位；最多有 10 个外围 I／O 处理器，它们可以与 CPU 并行地工作，每个 I／O 处理器有一个字长 12 位 ALU，这样，

$$\begin{aligned} T(\text{CDC 6600}) &= T(\text{中央处理器})\times T(\text{I／O 处理器}) \\ &= \langle 1, 1\times 10, 60\rangle \times \langle 10, 1, 12\rangle \end{aligned}$$

然而，Handler 分类又过于依赖机器，目前大家都普遍遵循 Flynn 分类法。

1.2 并行计算模型

计算模型是算法的实现基础。并行算法的设计严格地依赖其计算模型。对同一问题而言，完全可能有许多不同的并行算法以适应在不同的模型上对这一问题的求解。

关于并行计算模型已有人做了大量的研究工作，在这里我们不打算给各种并行计算模型做一个系统的总结，仅仅介绍一些公认的计算模型。

前面介绍的几种并行计算机，它们是并行算法的物质基础。但对算法的研究者而言，不能仅局限于某种具体并行机而研究并行算法，必须从算法角度，将各种并行机的基本特征加以理想化，抽象出所谓的并行计算模型，然后在此模型上研究和发展各种有效的并行算法。为此，我们在并行计算模型中对多处理器并行机作如下一些假定：

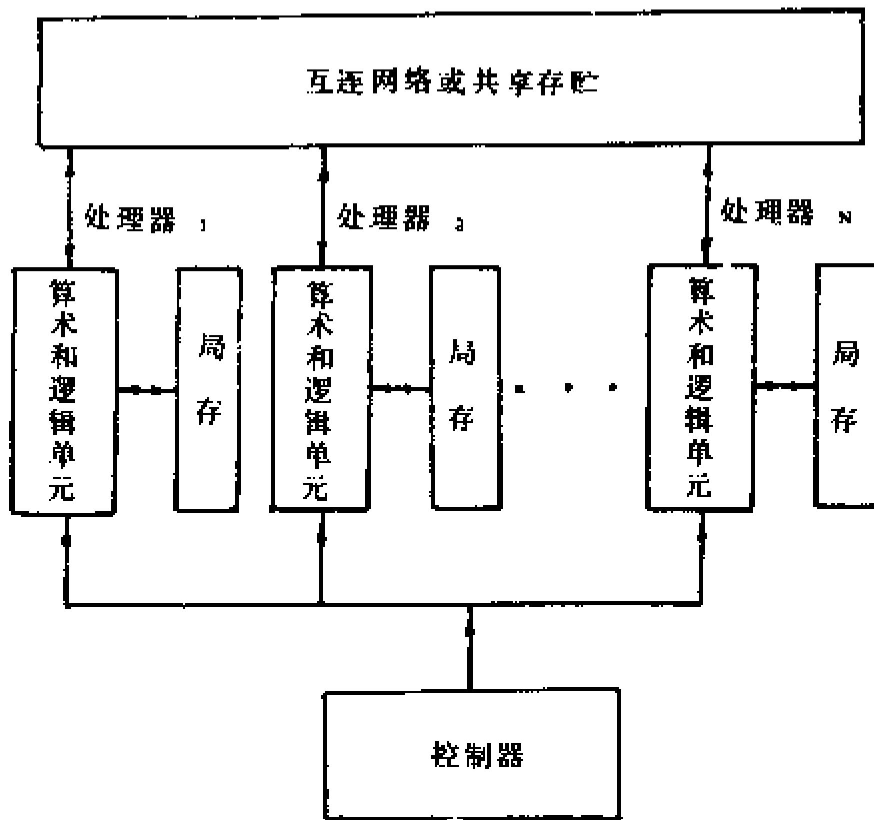


图 1.5　SIMD 计算模型

(1) **处理器数目假定**：无限和有限。前者是指算法中可使用随问题规模(大小或尺寸) n 成高阶多项式增长的处理器数；后者则是用 p 个处理器去求解问题($p\leqslant n$)；

(2) **时间假定**：每个处理器执行指令或数据操作的时间相同，且与处理器数目多少无关；

(3) **存取假定**：允许多个处理器同时读、写一个共享存贮单元，或不允许这样做；

(4) **指令集假定**：每个处理器具有普通计算机所拥有的指令集，如算术或逻辑运算、访内、输入／输出等指令。

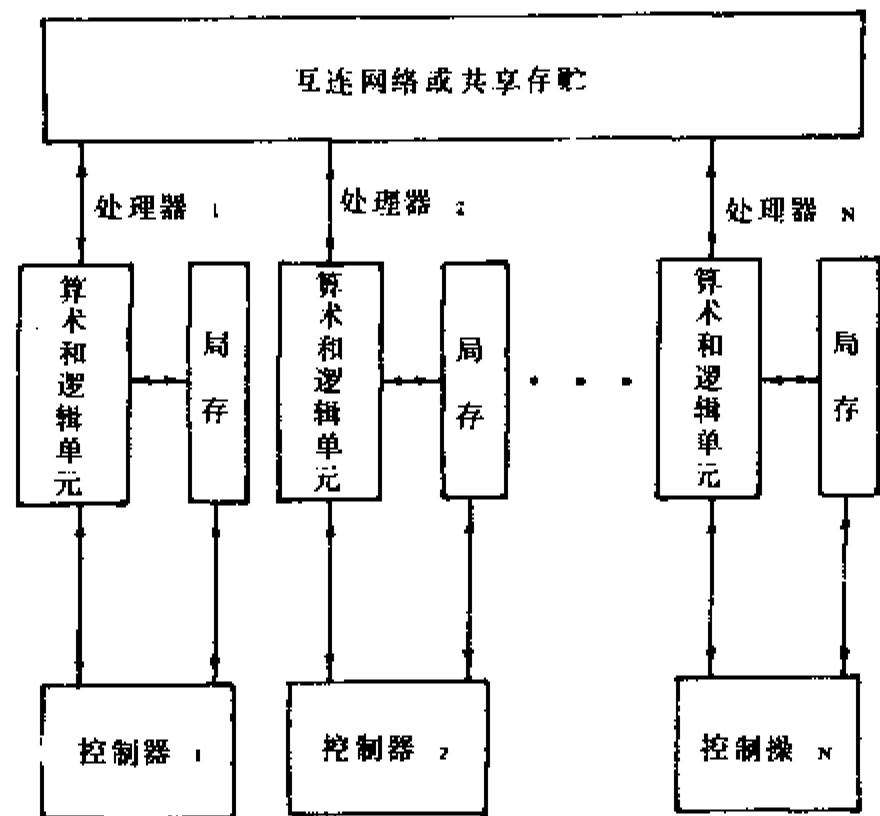


图 1.6　MIMD 计算模型

一般而言，并行计算模型可分为专用并行结构和通用并行结构。所谓**专用并行结构**是专门为有效解决某一类特定问题而设计的一种并行结构，如用于排序和选择的**比较器网络**(Comparator Network)[11]。基于 VLSI 技术的 **Systolic 阵列**[12]，这种模型是由 Kung 提出的，由于它具有结构规整，且以流水方式操作，很适合于数值计算。还有一种 VLSI 模型[13,14]，这种模型是指在一块芯片上装入成千上万个部件线路的计算模型，其中每个处理部件完成单一的逻辑功能，处理部件之间通过线路连接通讯，由于 VLSI 具有将算法渗透于硬件中的特性，近几年颇受人们重视。所谓**通用并行**

结构是为了解决广泛的应用问题而设计的一种并行结构。根据 Flynn 分类法，通用并行结构可分为 SIMD 及 MIMD 两大类。这两大类还可进一步细分为基于共享存贮的 SIMD 及 MIMD 计算模型和基于互连网络的 SIMD 及 MIMD 计算模型。它们分别如图 1.5 及图 1.6 所示。

1.2.1 SIMD 共享存贮模型

共享存贮模型是一种理想的计算模型，Fortune 等人[15]最初描述的共享存贮模型是：假定存在一个容量无限大的共享存贮器——**并行随机存取机器** PRAM(Parallel Random Access Machine)，同时有**有限**(Bound)或**无限**(Unbound)个功能相同的处理器，每个处理器拥有简单的算术运算和逻辑判断功能。在任何时刻，任何一个处理器均可通过共享存贮器的共享单元同其它任何处理器互相交换数据，但根据处理器对共享单元存取的不同约束条件又进一步分为：

(1) **不允许同时读和同时写**(Exclusive-Read and Exclusive-Write)。换句话说，每次仅允许一个处理器读或写某一个共享单元。这种计算模型简记为 SIMD-EREW PRAM；

(2) **允许同时读，但不允许同时写**(Concurrent-Read and Exclusive-Write)。换句话说，每次允许任意多个处理器同时读一个共享单元的内容，但每次仅允许一个处理器向某个共享单元写内容。这种计算模型简记为 SIMD-CREW PRAM；

(3) **允许同时读和写**(Concurrent-Read and Concurrent-Write)，即每次允许任意多个处理器同时读和同时写同一个共享存贮单元。这种计算模型简记为 SIMD-CRCW PRAM。

上述三种计算模型，根据它们对共享存贮单元的存取约束的强弱又分为最弱的计算模型 SIMD-EREW PRAM 和最强的计算模型 SIMD-CRCW PRAM。Snir 已证明了 SIMD-EREW PRAM 严格弱于 SIMD-CREW PRAM[16]。Cook 等人则又证明了 SIMD-CREW PRAM 严格弱于 SIMD-CRCW PRAM[17]。因此，若 T_M 表示一并行算法在一并行计算模型 M 上的时间，则

$$T_{EREW} \geqslant T_{CREW} \geqslant T_{CRCW}$$

在 SIMD-CRCW PRAM 上，允许许多处理器同时向一共享单元写内容，虽然这是不现实的。为了解决这种"写"引起的冲突。Kucera 曾提出三种解决"写冲突"的方法[18]，它们分别是：

(1) 仅写"1"的处理器写成功；

(2) 写入相同内容的处理器写成功；

(3) 优先权最高的处理器写成功。

而且他还证明了在以处理器为代价的前提下，这三种解决"写冲突"方法的时间复杂性是一致的。最近 Fich 等人更进一步分析了在处理器数及共享单元数目都受限制的情况下，这三种解决"写冲突"之间的关系[19]。

1979 年，Eckstein 曾采用二叉树方法来解决读写冲突问题[20]。为了解决读引起的冲突，只允许一个处理器从共享单元取内容，然后通过树向所有其它处理器广播此数据。为了解决写冲突，将树作为一种竞赛(Tournament)机构，确保仅有一个处理器写内容，这样，一个具有时

间复杂性 T 和空间复杂性 S 的、在 SIMD-CREW PRAM 或 SIMD-CRCW PRAM 上的算法，在 SIMD-EREW PRAM 上模拟实现时，其复杂性分别为：

$$T_{EREW}=O(T_{CREW}\cdot \log p)=O(T_{CRCW}\cdot \log p)$$

$$S_{EREW}=O(S_{CREW}\cdot p)=O(S_{CRCW}\cdot p)$$

其中 p 为处理器数目．可见，用二叉树解决读写冲突问题是费空间的．

1983 年，Vishkin 曾提出了另外一种解决读写冲突方法[21]．所得结果是：

$$T_{EREW}=O(T_{CREW}\cdot \log^2 p)=O(T_{CRCW}\cdot \log^2 p)$$

$$S_{EREW}=O(S_{CREW}+p)=O(S_{CRCW}+p)$$

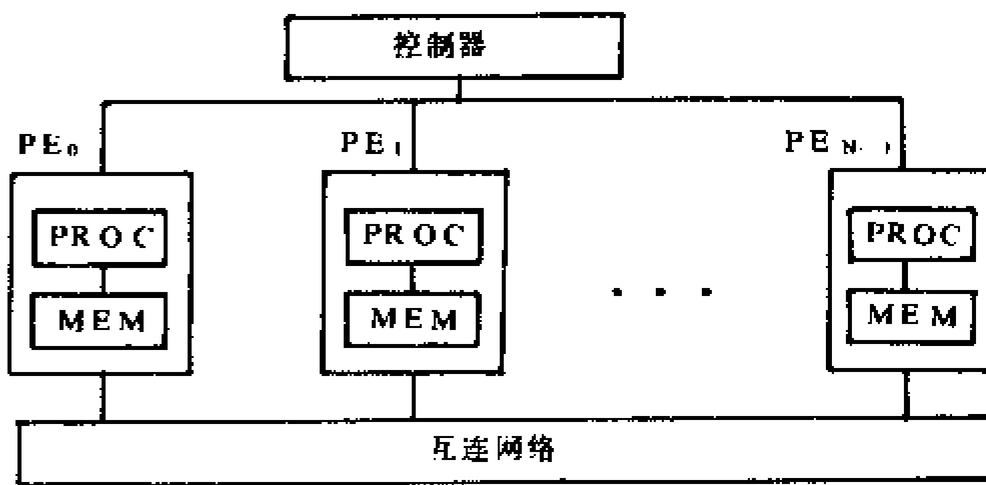


图 1.7 闺房式 SIMD 模型

显然，在空间上是省多了，但却增加了时间复杂性。总之，解决读写冲突，应很好地考虑时、空平衡。

在现实的各种并行计算机中，若不考虑处理器之间的通信开销几乎是不可能的，通信开销在算法中甚至占主导地位．而上述基于共享存贮的并行计算机是不存在的。人们之所以乐于采用它，是因为它抛开了各种具体的体系结构，使人们能够集中精力，从问题本身出发，研究和发展待求解问题本身固有的并行性以及并行计算的复杂性，从而使得并行算法的研究成为一项独立的活动。

1.2.2 SIMD 互连网络模型

SIMD 互连网络模型是一种比较切合实际的并行计算模型，许多实验用的和商品生产的并行计算机几乎都是基于互连网络结构的。而基于互连网络连接的 SIMD 机器又分为两大类。一类是处理器——处理器之间直接互连（又称为闺房式），另一类是处理器——存贮器之间直接互连（又称为舞厅式）。它们分别示于图 1.7、1.8 中。

闺房式适用于处理器和存贮器数目相等的场合；而舞厅式适用于存贮器数目多于处理器数目。但从算法设计者角度来讲，两者并无本质的差别。因此，我们仅限于讨论闺房式这一类。

下面我们列举一些常用到的互连网络模型。

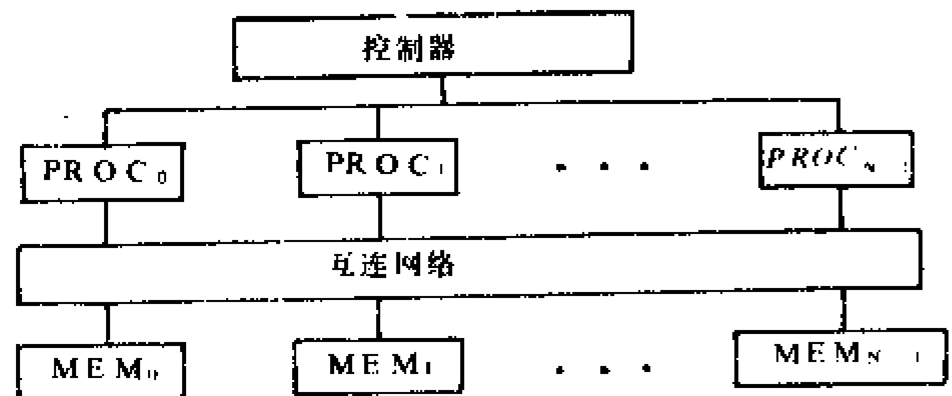


图 1.8 舞厅式 SIMD 模型

1. 一维线性模型

一维线性阵列(Linear Array)，其连接方式是所有并行机中处理器之间一种最简单、最基本的互连方式。其中每个处理器只与其左、右近邻相连（第一个和最后一个处理器例外），这种连接方式是 **Systolic** 结构中

的最基本形式。

若有 n 个处理器组成一维线性阵列，处理器编号依次为 0，1，2，…，$n-1$，则线性阵列的连接函数 LC 定义如下：

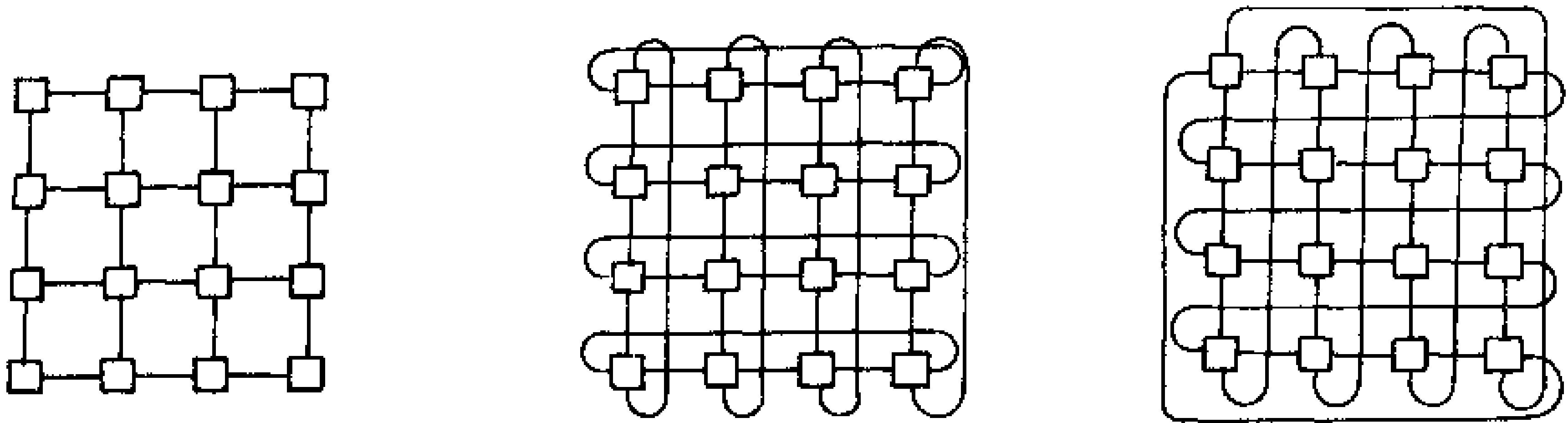
$$LC_{-1}(i)=i-1 \quad 1\leqslant i\leqslant n-1$$
$$LC_{+1}(i)=i-1 \quad 0\leqslant i\leqslant n-2$$

其中 i 是对应处理器编号，LC_{-1} 表示左连接，LC_{+1} 表示右连接。

2. 网孔模型

首先我们介绍最简单的二维网孔连接(2D Mesh-Connected)模型，然后再给出一般的 $q(\geqslant 2)$维网孔连接模型。

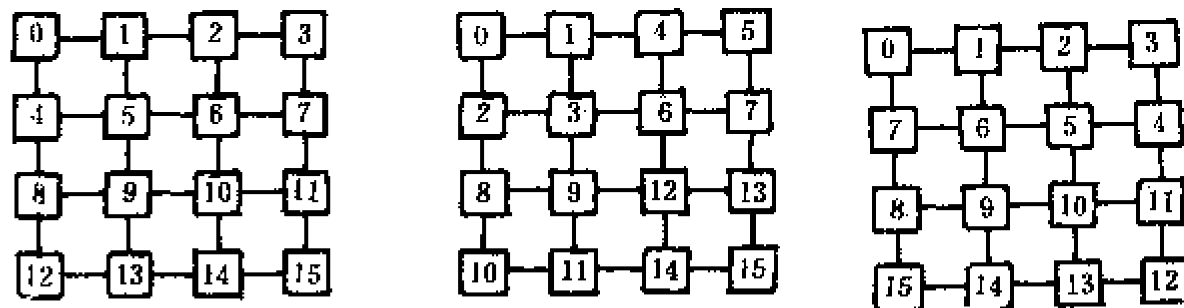
在二维网孔连接处理器阵列中，处理器按二维阵列形式排列，每个处理器仅与四个相邻处理器（若存在的话）有线互连。如图 1.9(a)所示。二维网孔阵列有许多变种，例如边界处理器也有线连接的两种方式，如图 1.9(b)及(c)所示。



(a) 不带绕回连接的网孔　(b) 相同行或列的处理器之间带绕回连接的网孔　(c) 带环形绕回连接的网孔

图 1.9　二维网孔连接

在二维网孔上处理器的编号方式常见的行主编号、洗牌编号以及蛇形编号，它们分别图示于图 1.10 的(a)、(b)及(c)中。



(a) 行主编号　(b) 洗牌行主编号　(c) 蛇形行主编号

图 1.10　处理器三种编号方式

对于有多个处理器组成的二维网孔阵列。若边界处理器的互连方式和图 1.10(c)相同，而处理器编号以行主方式进行，则处理器之间的互连函数 MC 为：

$$MC_{-1}(i)=(i-1) \bmod n$$
$$MC_{+1}(i)=(i+1) \bmod n$$
$$MC_{-\sqrt{n}}(i)=(i-\sqrt{n}) \bmod n$$

$$MC_{+\sqrt{n}}(i)=(i+\sqrt{n}) \bmod n \qquad (0\leq i<n)$$

其中：i 是处理器编号，MC_{-1} 及 MC_{+1} 表示 i 的行连接，$MC_{-\sqrt{n}}$ 及 $MC_{+\sqrt{n}}$ 表示 i 的列连接。图 1.11 给出 $n=16$ 的二维网孔的互连方式。

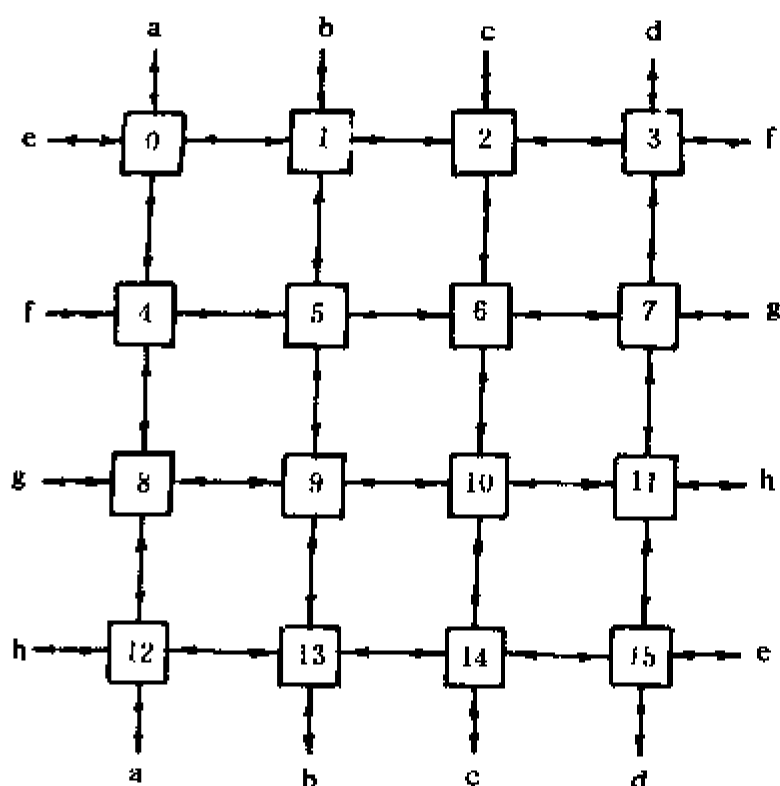


图 1.11　$n=16$ 的二维网孔

在二维网孔模型上，已设计出许多有效的排序、选择、矩阵运算以及图论等方面的并行算法。但网孔结构的通信功能较差，在最坏情况下，任意两处理器之间的信息交换步至少需要 $\sqrt{n}-1$ 步[23]。

q—维网孔模型是二维网孔模型的推广。假定有 n 台处理器，这 n 台处理器置于 q—维晶格空间的格点 $(i_1, i_2, \cdots, i_q)$ 上，i_j 是其中一个正整数，$1\leq j\leq q$。位于格点 $(i_1, i_2, \cdots, i_q)$ 上的处理器 P_x 与位于格点 $(j_1, \cdots, j_q)$ 上的处理器 P_y 有线互连，当且仅当它们之间的距离 $d(P_x, P_y)=1$，其中

$$d(P_x, P_y)=\sum_{k=1}^{q}|i_k-j_k|$$

q—维网孔模型在证明网孔上实现的并行算法的下界时常常用到。

3. 树机模型

二叉树是大家非常熟悉的一种数据结构。除了根结点外，每个非叶结点都与父结点和两个儿子结点有线连接。若一棵满二叉树有 d 级（编号自根至叶为 0 到 $d-1$），则共有 $n=2^d-1$ 个结点。图 1.12 示出了 $d=4$ 的一棵满二叉树机。其中根结点和叶结点具有 I/O 操作功能。树机的典型工作方式是：叶结点对数据进行并行计算，而内部结点仅负责叶结点间的通信及简单的逻辑运算。为此，Kung 等人曾提出叶结点是 SIMD 计算机，而非叶结点是简单 Systolic 比较器的树机模型，如图 1.13 所示。

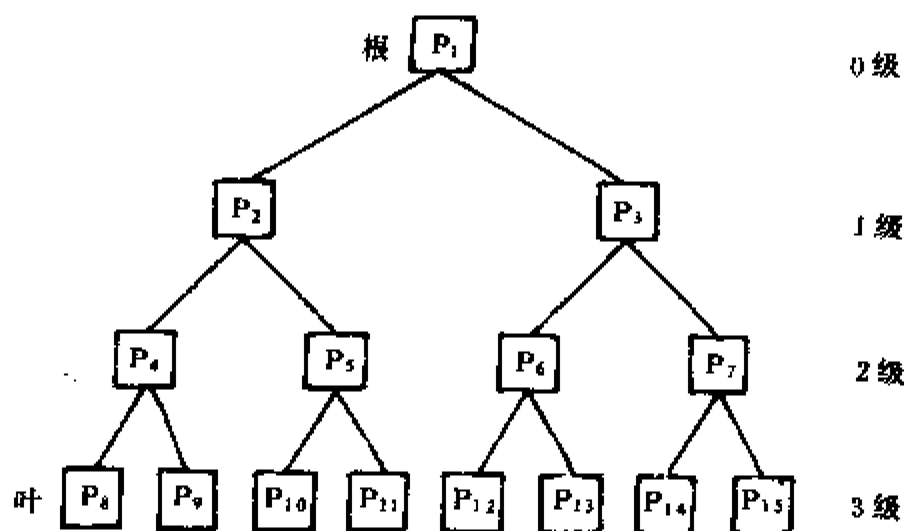


图 1.12　四级二叉树机

由于树根结点往往成为通信的瓶颈，为此有人建议在同一层结点之间也使用线连接起来，这就是所谓 X—树[22]，如图 1.14 所示。显然，X—树明显的改善了树机的通信功能。

4. 超立方及立方环模型

一个由 $n=2^q$ 个处理器组成的 q— **维超立方计算模型**，处理器 i 与处理器 j 有线

连接当且仅当 i 与 j 的二进制表示仅一位不同，$0 \leqslant i,\ j < n$.

图 1.15 给出了 $q=4$ 的 4 维超立方。若处理器编号 i 的二进制表示 $i_{q-1}i_{q-2}\cdots i_1 i_0$，则表的互连函数 CC 为:

$$CC_k(i_{q-1}i_{q-2}\cdots i_{k+1}i_k i_{k-1}\cdots i_0)=i_{q-1}i_{q-2}\cdots i_{k+1}\bar{i}_k i_{k-1}\cdots i_0$$

其中 $\bar{i}_k = 1 - i_k$，$0 \leqslant k < q$.

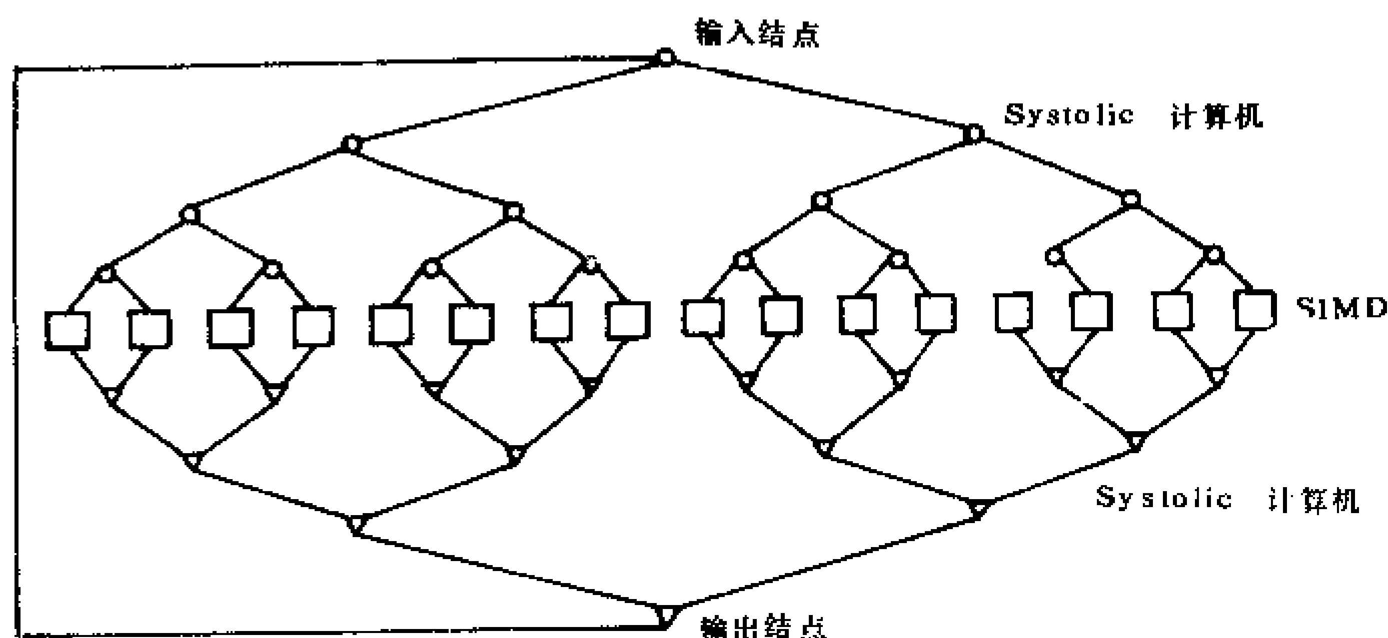


图 1.13 树状并行计算机的体系结构

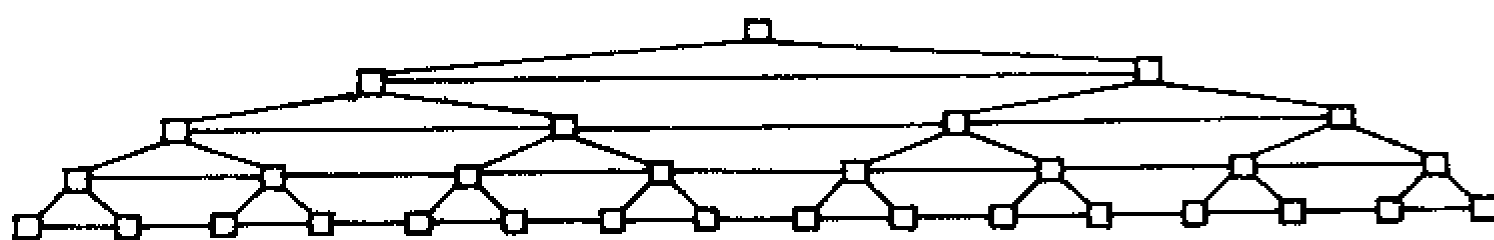


图 1.14 $n=2^5-1$ 的 X-树

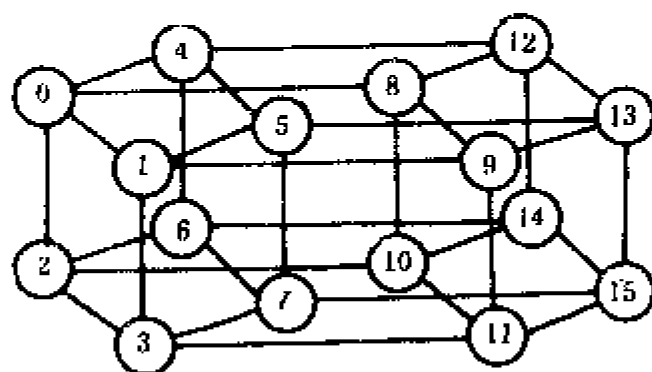


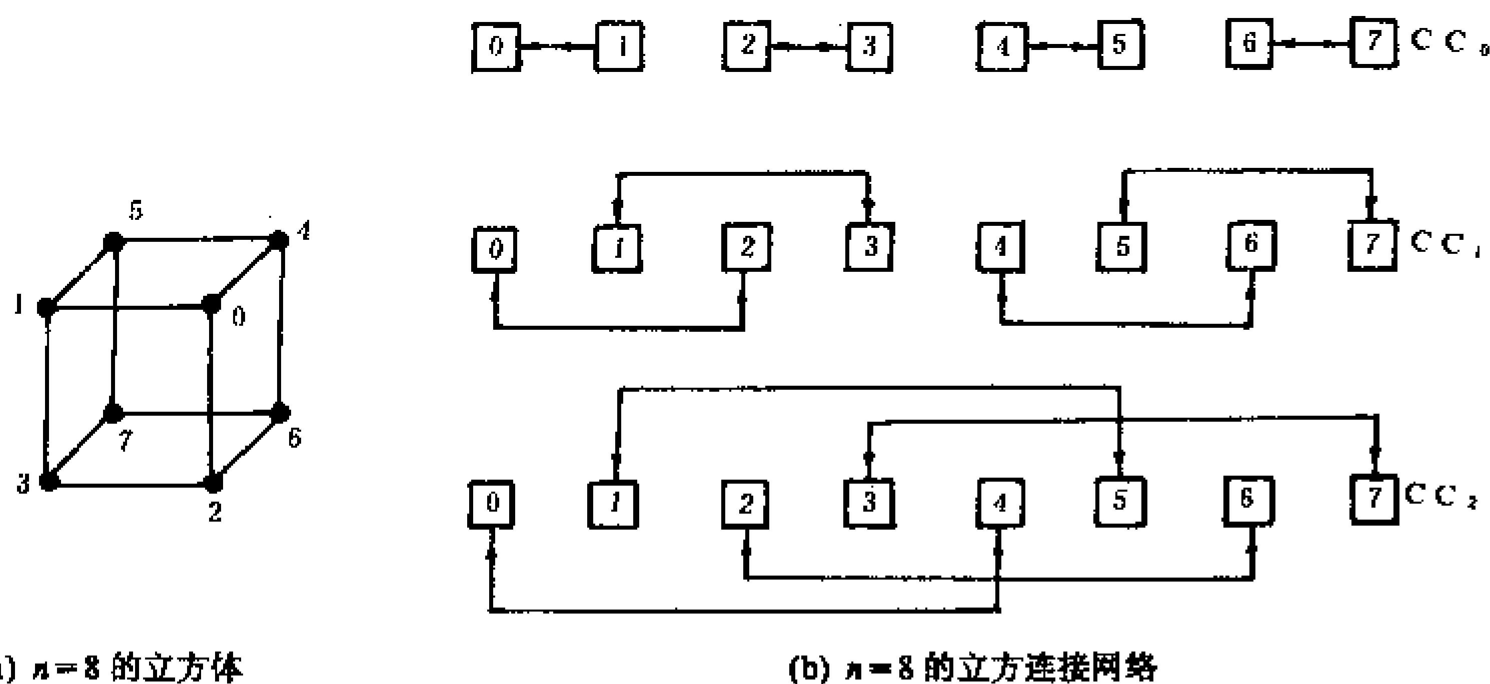
图 1.15 4-维超立方

$n=8$ 的立方连接网络（单级）示连 1.16，其中 CC_0 将地址第 0 位不同的那大理器相连；CC_1 将地址第 1 位不同的那址理器相连；CC_2 将地址第 2 位不同的那址理器相连。实际上，三种连接函数合在一址组成一个立方体。

Siegel 的研究表明：超立方模型是一连用的计算模型，它可以模拟很多其它形式大连网络[23]。但在一般情况下，每个处理器将同 $\lceil \log n \rceil$ 个处理器连接。当问题规的大时，从制造技术上讲是不可接受的，因而从某种意义上讲限制了超立方模型更可的应用。

1981 年 Preparata 等人建议了超立方模型的一种变种——立方环(Cube-Connected cles)，简记为 CCC[24]。它结合了环网和超立方连接的优点，使得每个处理器仅与三个

它处理器互连. 这样，处理器之间的连线数目与问题规模无关，这一点对 VLSI 的实现是极其重要的.



(a) $n=8$ 的立方体　　(b) $n=8$ 的立方连接网络

图 1.16　立方连接网络

CCC 构造如下：假定 $n=2^q$ 个处理器，其中 $q=r+2^r$，r 取满足 $r+2^r \geqslant q$ 的最小整数，则 CCC 的环内线连接与环之间的线连接定义如下：

环内：编号为 j 的处理器同编号为 $(j+1)$ **mod** 2^r 的处理器用线连接起来，$0 \leqslant j < n$。

环同：编号为 $(l2^r+j)$ 的处理器同编号为 $(l\cdot 2^r+j+(1-2l_j 2^j)$ 处理器互连，$0 \leqslant l \leqslant 2^{2^r}$，$0 \leqslant j < 2^r$.

其中：l_j 是 l 的二进制数表示的第 j 位（最右边为第 0 位），即 $l_j=0$ 或 1。$(1-2l_j)2^j$ 表示将 l 的二进制第 j 位取反。可见，每个环是由 2^r 个连线连接而成的。每个环下标为 l（共有 2^{q-r} 个环）。若将每个环上的处理器都视为以此环为代表的超顶点，则环之间的连线构成一个 2^{q-r} 维超立方。图 1.17 给出了一个 $n=32(q=5,\ r=2)$的 CCC 网络。它共有 $2^{5-2}=8$ 个环，每个环内共有 $2^r=4$ 处理器。

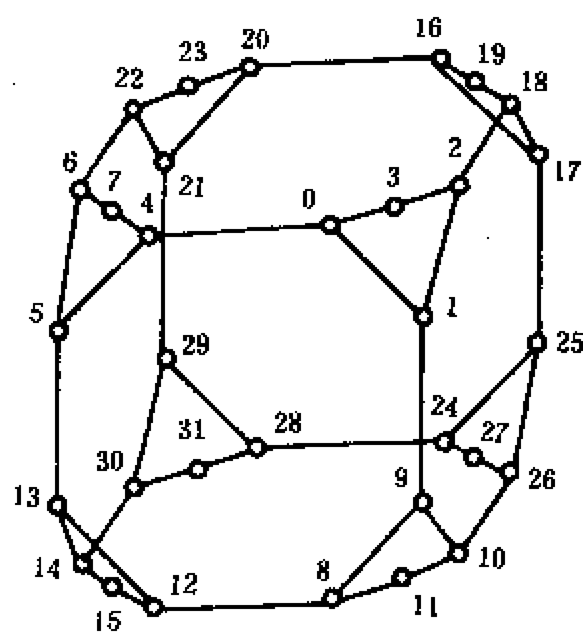


图 1.17　$n=32$ 的 CCC 结构

每个处理器编号的二进制表示有 q 位. 这 q 位地址可分解为一对整数(l,p)表示，其中 l 占 $k-r$ 二进制位，p 占 r 个二进制位。若一处理器编号为 i，则 $l2^r+p=i$，$0 \leqslant i < n$。每个处理器含有三个与其它处理器的连接函数，它们分别是 F(Forward)，B(Backword)和 L(Lateral)，定义如下：

环内：$F(l,p)$ 连向 $B(l,(p+1)$ **mod** $2^r)$

　　　$B(l,p)$ 连向 $F(l,(p-1)$ **mod** $2^r)$

环间：$L(l,p)$ 连向 $F(l+\varepsilon 2^{p},p)$，$\varepsilon=1-2i_{p}$

Galil 等人证明了 CCC 是一种非常有效的通用计算模型[25]．它可以有效地模拟许将其它互连网络．

5. 洗牌一交换模型

洗牌一交换(Shuffle-Exchange)是另一类非常有用的互连结构．洗牌的名字来源于"牌牌"游戏，即将一叠牌对分两两半，然后依次从两半中各一一张相互叠加起来，这种方也也叫**均匀洗牌(Perfect Shuffle)**，有时也称之为**混洗**．

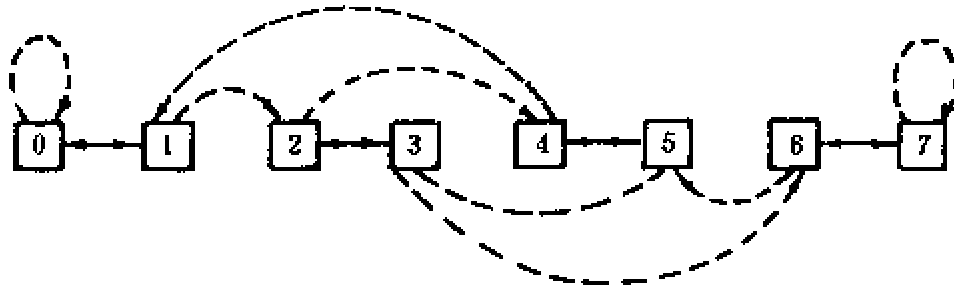


图 1.18 $n=8$ 的洗牌交换网络

洗牌函数作为一种连接函数往往不充分，所以常将它与交换函数配合使用，形成了"洗牌一交换"网络．有时也简称洗牌网络．

简单地讲，若有 p 个处理器组成一个洗牌模型，那么处理器间连接方式为：编号为的处理器同编号为$(2i-1)$的处理器互连；编号为$(i+p/2)$的处理器同编号为 $2i$ 的处理器连；编号为$(2i-1)$的处理器与编号为 $2i$ 的处理器互连($i=1,\ 2,\ \cdots,\ p/2$)．对于一般由个处理器组成的洗牌网络而言，设处理器 i 的二进制表示为 $i_{q-1}i_{q-2}\cdots i_2i_1i_0$，则洗牌及交函数如下：

$$\text{Shuffle}(i_{q-1}i_{q-2}\cdots i_2i_1i_0)=i_{q-2}i_{q-1}\cdots i_1i_0i_{q-1}$$

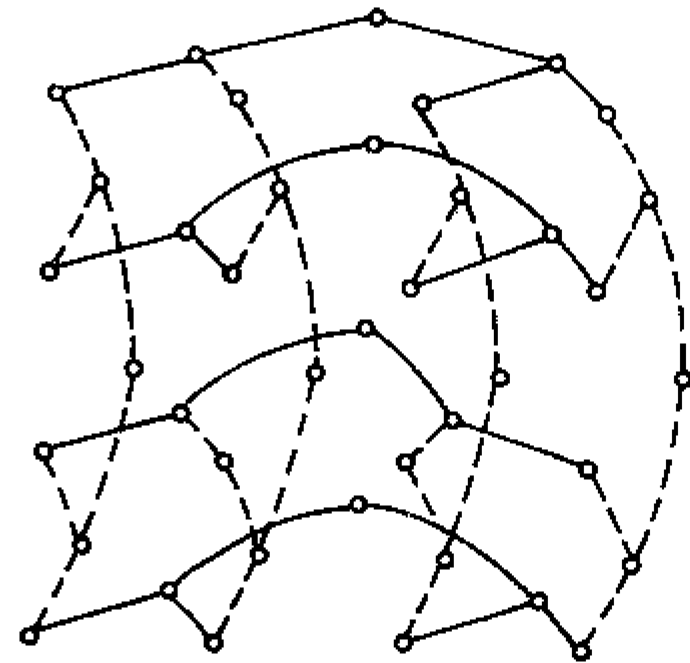
$$\text{Unshuffle}(i_{q-1}i_{q-2}\cdots i_2i_1i_0)=i_0i_{q-1}i_{q-2}\cdots i_2i_1$$

$$\text{Exchange}(i_{q-1}i_{q-2}\cdots i_2i_1i_0)=i_{q-1}i_{q-2}i_{q-3}\cdots i_2i_1\bar{i}_0$$

其中$\bar{i}_b=1-i_b$，$0\leqslant b<q$．显然，洗牌的功能实际上循环左移一位；逆洗牌功能实际上是循环右移一位；而交换的功能则是将地址相邻的两处理器数据进行交换．

图 1.18 给出了 $n=8$ 的洗牌网络，其中实线表示 Exchange 函数，而虚线表示 Shuffle 函数．

洗牌交换网络，由于它能方便地模拟超立方网络的功能，且每个处理器仅同其它三个处理器互连，因而受到计算机工作者广泛注意，但由于它连线的不规整性，给布局带来了很大的困难．


图 1.19　4×4 的树图

6. 树网模型

从通信角度来讲，树结构明显优于网孔结构，但网孔结构有它的优点，将这两者优起来，发展成为一种新型互连网络模型——**树网(Mesh-of-Tree)**模型．树网的构造，简来讲，将所有处理器按二维阵列形式安排，然后每行及每列的处理器再按完全二叉树网

连接起来，(它们分别作为二叉树的叶结点)。如图 1.19 所示。

这样，可通过相应的行树及列树进行通信，由于这种结构既具有良好的通信功能，又继承网孔上很多的有效算法，同时它也非常适合于 VLSI 集成化，故在这种结构上已发展了许多有效的并行算法。

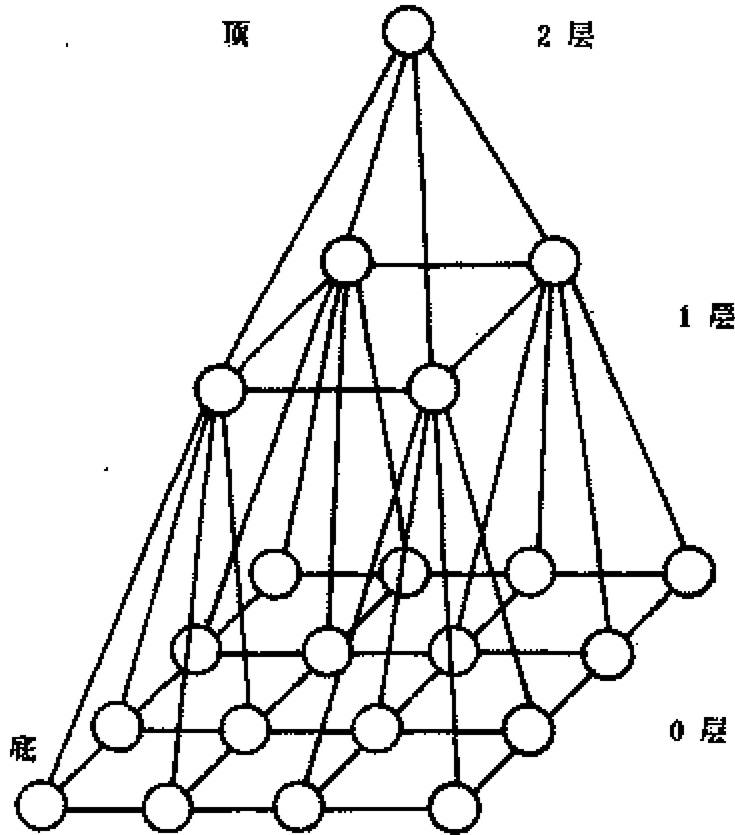


图 1.20　$p=16$ 的金字塔结构

7. 金字塔模型

金字塔(Pyramid)模型是由树与网孔结合而成的。这种结构在图象处理等领域颇为有用。一个由 $O(p)$个处理器组成的金字塔，是一棵高度为 $\log_4 p$ 的完全四叉树，而四叉树的每层处理器又按二维网孔方式互连。其中 $p=k^2$ 是二维网孔形式的金字塔塔底。金字塔层数由底向上从 0 开始编号，且塔顶只有一个处理器，第 i 层的一个处理器至多与其它九个处理器互连。它们分别是：第 $i+1$ 层的父结点；第 $i-1$ 层的四个儿子结点以及同一层的四个相邻结点（若它们分别存在的话）。因此一个高为 $\log_4 p$ 的金字塔共有$(4p-1)/3$ 个处理器，图 1.20 是 $p=16$ 的金字塔。

8. 蝶形模型

蝶形(Butterfly)结构是与 FFT 密切相关的一种互连结构。从拓扑结构上讲，它与多级立方网络，榕树网络和归并网络颇为相似。如图 1.21 所示。它共有$(k+1)2^k$ 个结点，组成$(k+1)$行(第 0 行至第 k 行)，其中第 0 行和第 k 行可视为等同，且每行有 $n=2^k$ 个结点。若令 $PE_{r,i}$ $(0\leqslant r\leqslant k,\ 0\leqslant i\leqslant n)$ 代表处于第 r 行第 i 列的处理器，那么在第 $r(r>0)$行上的处理器 $PE_{r,i}$ 将连向 $PE_{r-1,j}$，其中 $j=i$ 或 j与 i 的二进制表示仅第 r 位不同(从左数)。

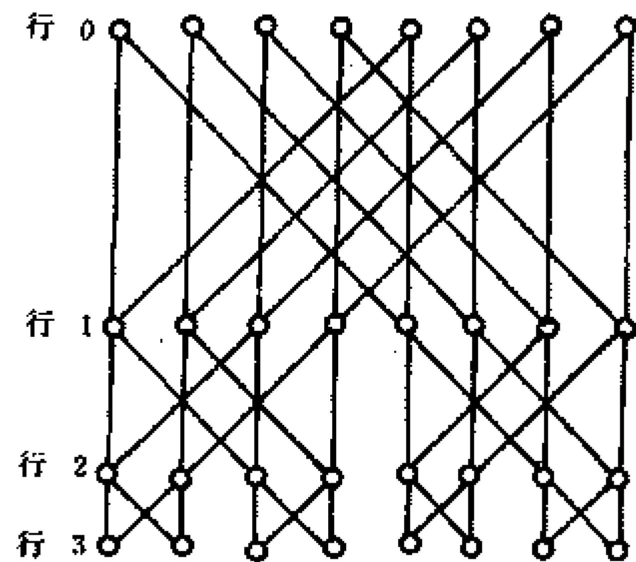


图 1.21　$k=3$，$n=8$ 的蝶形连接

在蝶形结构上已研究出一类并行排序及 FFT 算法[14]。

1.2.3　MIMD 并行计算模型

1. MIMD 共享存贮模型

SIMD 模型同 MIMD 模型的一个显著差别，前者有一个全局控制器，而后者没有全局控制器，在 MIMD 共享存贮模型中，假定存在一个无限大的共享存贮器，每个处理器各自完成自己的子任务，但为了完成整个任务，各处理器之间需要交换数据，这可由同步机制实现，也就是说，通过共享存贮变量实现各处理器之间的通信。在这种模型上开发的

算法称之为**异步并行算法**(Asychronous Parallel Algorithm)。

因 MIMD 共享存贮模型同现实相差甚远，目前在它上面开发的并行算法非常有限。

2. MIMD 异步通信模型

一个基于**异步通信的分布式计算模型**可以抽象为一个无向图 $G(V,E)$，其中顶点集对对应处理机集合，边集合 E 对应处理机间的双向通信链集合。每个处理机都有一个唯的的编号。而且每个处理机只知道邻接处理机（有线直接互连的处理机）这一局部知识。理理机之间的通信是通过发送和接收消息完成的。处理机之间不存在共享存贮器。在算法行行期间，假定处理机及通信链均不能出现故障(Fail)。每个处理机除了执行自己的计算务务外，还向邻接处理机发送消息以及接收、处理来自邻居的消息，并假定这些操作所需时时间同处理机间通信时间比较起来可以忽略不计。每个处理机发送给邻接处理机的消息有有限的、不确定的时间内到达，且在一条通信线的同一方向上的消息到达目标的次序服机"先进先出(FIFO)"规则。

在基于异步通信的分布式计算模型上设计的算法称之为**分布式算法**(Distributs Algorithm)，其主要衡量标准是算法的通信复杂性以及通信所花费的时间。

1.3 小　　结

本章概述了并行算法的一些基础知识。主要内容包括：并行计算机的发展简史和现的的几种著名的并行计算机；以及并行计算机的分类法。着重介绍了 SIMD 机器上两种型型的计算模型：(1)基于**共享存贮**的模型：有 SIMD-EREW PRAM、SIMD-CREPR PRAM 以及 SIMD-CRCW　PRAM，并对它们之间的相互模拟进行了一些讨论；(2)于于**互连网络**的模型：有一维线性阵列、网孔、树机、超立方及立方环、洗牌一交换、网网、金字塔和蝶形网络等，最后扼要介绍了 **MIMD 机器**上基于**共享存贮模型**及基于**异通通信的互连网络模型**。本章介绍的模型是后面各章要介绍的并行算法的物质基础。另了牌本章还介绍了并行算法中的一些常用的基本概念。

参　考　文　献

[1] 陈国良．并行算法——排序和选择，合肥，中国科学技术大学出版社，1990

[2] 梁维发．图论问题的并行算法，中国科学技术大学硕士论文，1989

[3] Hwang　K，Briggs　F　A. *Computer Architecture and Parallel Processing*，McGraw-Hill, NY，19

[4] 金兰，王鼎兴，沈美明．并行处理计算机结构，国防工业出版社，1982

[5] Bouknight　W　J　et　al. The Illiac IV System，*Proc. of the IEEE*, **60**(4)，1972，369-388

[6] Control　Data　Corporation．Control Data STAR-100，*Computer*，1972

[7] Wulf W A，Bell C G. C.mmp—A Multi-Mini-Processor，*AFIPS Proc. FJCC*, **41**，1972，765-777

[8] Swan　R　J, Fuller　S　H，Siewiorek　D　P. C*m—A Molular, Multi-Microprocessor. *AFIPS Pr NCC*, **46**，1977，637-644

[9] Flynn M J. Very High–Speed Computing System. *Proc. of the IEEE*, **54**(2), 1966, 1901–1909

[10] Handler W. The Impact of Classification Schemes on Computer Architecture, *Proc. 1977 Int'l Conf. on Parallel Processing*, New York, 1977, 7–15

[11] Batcher K E. Sorting Networks and Their Aplications. *AFIPS Proc. SJCC*, **32**, 1968, 307–314

[12] Kung H T. Why Systolic Architecture? *IEEE Computer*, **15**, 1982, 37–46

[13] Thompson C D. Area–Time Complexity of VISI, *Proc. 11th Annu. ACM Sympo. On Theory of Computing*, 1979, 81–88

[14] Ullman J D. *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD., 1984

[15] Forture S, Wyllie J. Parallelism in Random Access Machines. *Proc. 10th Annu. ACM Sympo. on Theory of Computing*, 1978, 114–118

[16] Snir M. On Parallel Searching, *SIAM J. Comput.*, 14, 1985, 688–708

[17] Cook S A, Dwork C, Reischuk R. Upper and Lower Time Bound For Parallel Randon Access Machines Without Simultaneous Writes. *SIAM J. Comput.*, **15**, 1986, 87–99

[18] Kucera L. Parallel Computation and Conflicts in Memory Access. *Inform. Proc. Lett.*, **14**, 1982, 93–96

[19] Fich F E, Ragde P, Wigderson A. Relations Between Concurrent Write Models of Parallel Computation, *SIAM J. Comput.*, **17**, 1988, 606–627

[20] Eckstein D M. BFS and Biconnectivity, TR 79–11, Dept. of Computer Sci., Iowa State Univ., of Sci. and Tech., Ames. Iowa, 1979

[21] Vishkin U. Implementation of Simultaneous Memory Access in Models that Forbit It. *J. Algorithms*, **4**(1), 1983, 45–50

[22] Sequin C H, Despain A M, Patterson D A. Communication in X–tree, a Modular Multiprocessor System, *ACM* 78 Proc. Washington D.C., 1978

[23] Siegel H J. *Interconnection Networks for Large–Scale Parallel Processing, Theory and Case Studies*, Lexington Books, Lexington, Mass., 1984

[24] Preparata F P, Vuillenmin J. The Cube–Connected Cycles: Versatile Network for Parallel Computation, *Commu. ACM*, **24**, 1981, 300–309

[25] Galil Z, Paul W J. An Efficient General–Purpose Parallel Computer. *J. ACM*, **30**(2), 1983, 360–387

第二章　并行算法的度量与设计技术

2.1　并行算法的基本概念

算法是解题方法的精确描述，它由一组有穷规则组成，这些规则规定了解决某一特定类型问题的一系列运算。所谓**并行算法**，简单地讲，就是适合于在各种并行计算机上求解问题的算法。

1980 年 Kung 曾经给并行算法下了一个精确的定义[1]，即：**并行算法是一些可同时执行的诸进程的集合，这些进程相互作用和协调动作，从而达到对给定问题的求解。**

并行算法可以从不同的角度加以分类：数值计算和非数值计算的；同步的、异步的和分布式的；SIMD 机器上的、MIMD 机器上的和 VLSI 模型上的等等。

所谓**数值计算**（Numerical Computation），是指基于**代数关系运算**的一类计算问题，诸如矩阵运算、多项式求值、解线性方程组等。它基本上属于数值分析(对以数字形式表示的问题求数值解)的范畴。

所谓**非数值计算**（Non−numerical Computation），是指基于**比较关系运算**的一类计算问题，诸如排序、选择、搜索、匹配以及图论等方面的计算问题。基本上是属于符号（如字符、数字、图象等）处理的范畴。

因此，我们把面向数值计算和非数值计算的并行算法分别称之为数值计算的并行算法和非数值计算的并行算法。

同步算法（Synchronized Algorithm）是指某些进程必须等待别的进程的一类并行算法。因为一个进程的执行，依赖于输入数据以及系统中断，所以全部进程均必须同步在一个给定的时钟，以等待最慢的进程。

一般地讲，运行在 SIMD 机器模型上的并行算法称为**同步并行算法**（Synchronous Parallel Algorithm）。

异步算法（Asychronized Algorithm）是指诸进程的执行一般不必相互等待的一类并行算法。在这种情况下，进程通信是通过动态地读取（修改）共享存贮器的全局变量来实现。

一般而言，运行在 MIMD 共享存贮模型上的算法称之为**异步并行算法**（Asychronous Parallel Algorithm）。

所谓**分布式算法**（Distributed Algorithm），是指基于异步通信模型上设计的一类算法，这一类算法的典型特征是：通过消息通信来协同整个问题的求解。

VLSI 并行算法（VLSI Parallel Algorithm）是指在 VLSI 计算模型上开发的一类并行算法。

2.2　MIMD 机器上的算法

在 SIMD 机器模型上的算法是同步的并行算法。在 MIMD 机器模型上的算法一般又

可分为三类：流水线（Pipelining）算法、划分（Partitioned）算法以及张驰(Relaxed)算法。

1. 流水线算法

一个流水线算法就是一组有序的程序段，其中前一段的输出将是下一段的输入。算法的输入就是第一段的输入；算法的输出就是最后一段的输出。象通常生产流水线一样，所有的段必须以相同的速率产生结果，否则最慢的段将成为瓶颈口。有的作者称此类算法是**宏流水线算法**(Macropipelining Algorithm)。

脉动算法（又称心动算法，Systolic Algorithm）也是一类特殊的流水线算法。它有三个基本属性：数据的流动有节奏、有规律；数据的流向可以多于一个方向；各段执行的运算基本一致。在一个 **Systolic** 算法中，要求产生数据的进程与消耗数据的进程之间隐含着同步的机制。

2. 划分算法

划分(Partitioning)是一种分治思想：将一个大问题划分成规模大致相等的子问题；然后在各个处理器上单独求解这些子问题；最终将所有解合并起来即给出原问题的解。将各处理器的解合并在一起就隐含着同步，所以划分算法也称为同步算法。

划分算法分为两类：**预调度算法**（Preschedule Algorithm）和**自调度算法**(Self-Scheduled Algorithm)。前者在编译时分配任务，后者在运行时动态指派任务。当一个计算是由大量子任务完成，而且每一个子任务均知道其复杂度时，适宜于使用预调度算法。在自调度算法中，必须保持待完成的任务表，一旦一个进程闲时，另一任务就从等待表中移出。进程调度本身也作为程序运行，因此称之为自调度。

3. 张弛算法

进程执行的进展不需同步而等待的算法称之为**张驰算法**。所有进程都可以朝同一目标(如划分算法那样)，或可以有某种特殊目的（如流水线算法那样）工作之，但决不存在一个处理器停下等待另一个处理器提供数据。张驰算法的特征是：诸处理器能够使用最新有效数据。正是因为张驰算法的不确定行为，致使预言其性能变得困难。

2.3 并行算法的度量标准

2.3.1 算法复杂性的基本概念

一个算法的**复杂性**（Complexity），也叫**复杂度**，是指它含有的工作量。例如，在串行算法（又称顺序算法）中，算法的复杂性是指算法的运算量（即时间步）与存贮量（存贮空间）。它们都是求解问题的规模函数。当问题的规模 n 趋向无穷大时，我们就得到算法复杂性的渐近表示。一般而言，我们关心的是 n 很大时的复杂性，这时它与渐近复杂性相差不多，在算法分析时，往往对这两者不予区分。

在对算法复杂性进行分析时，我们常使用**上界**（Upper Bound）、**下界**（Lower Bound）以及**精确界**（Tightly Bound）概念。下面我们就精确地定义这些基本概念。

令 $f(n)$ 和 $g(n)$ 是定义在自然数集合 N 上的两个函数，如果存在两个正常数 c 及

n_0，使得对所有 $n > n_0$ 均有

$$f(n) \leqslant c \cdot g(n)$$

则称 $g(n)$ 是 $f(n)$ 的一个**上界**，记作 $f(n) = O(g(n))$。注意：在求 $f(n)$ 的上界时总是试图求出最小的 $g(n)$，使得 $f(n) = O(g(n))$。

如果存在两个正常数 c 及 n_0，使得对所有的 $n > n_0$，均有

$$f(n) \geqslant c \cdot g(n)$$

则 $g(n)$ 是 $f(n)$ 的一个**下界**，记作 $f(n) = \Omega(g(n))$。同样的，在求 $f(n)$ 的下界时，总是试图求出最大的 $g(n)$，使得 $f(n) = \Omega(g(n))$。

如果存在正常数 c_1，c_2 及 n_0，使得对所有的 $n > n_0$，均有

$$c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n)$$

则 $g(n)$ 是 $f(n)$ 的**精确界**，也称**紧致界**，记作 $f(n) = \Theta(g(n))$。注意：一个算法的复杂性 $f(n) = \Theta(g(n))$，就意味着此算法在最好及最坏情况下的复杂性在一个常量因子范围内是相同的。

从算法复杂性角度，我们又可把算法分成两大类：凡是复杂性函数上界是多项式界的算法称之为**多项式复杂性算法**(Polynomial Complexity Algorithm)；而复杂性函数上界是指数界的算法称之为**指数复杂性算法**(Exponential Complexity Algorithm)。

几种常见的多项式与指数复杂性函数关系如下：

$$O(1) < O(\log n)^{①} < O(n) < (O(n\log n) < O(n^2) < O(n^3)$$

$$O(2^n) < O(n!) < O(n^n)$$

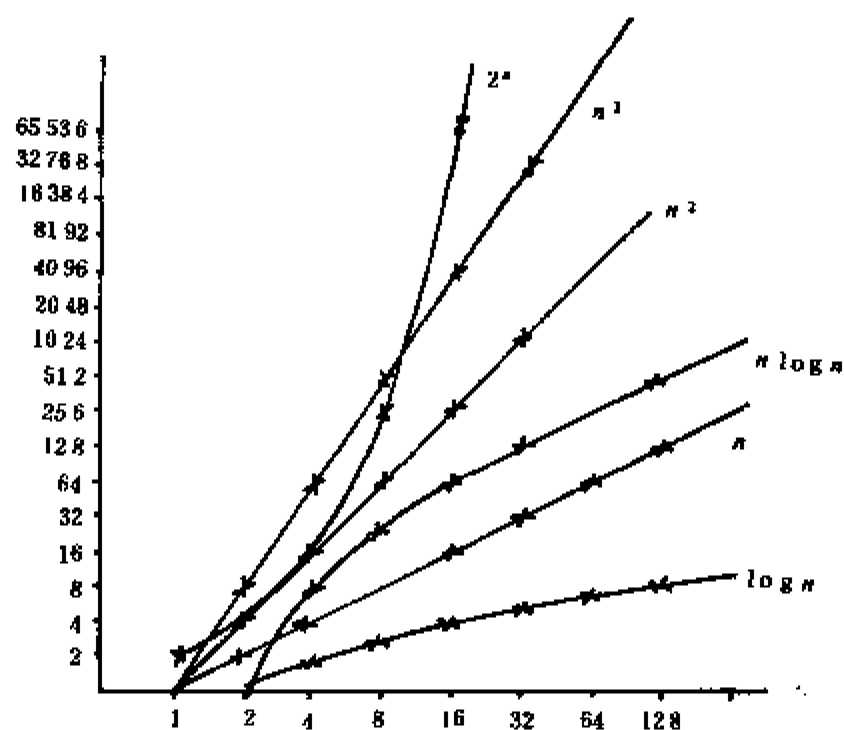


图 2.1 几种常见的函数曲线

图 2.1 给出了几种函数曲线之间相互关系。

2.3.2 并行算法的复杂性度量

算法可用不同的标准度量，但我们主要关心的是算法与求解问题规模之间的关系。对于一个给定的具有规模 n 的问题，通常有各种可能的输入集合。在算法分析时，对算法的所有输入分析其平均复杂性称之为**期望复杂性**(Expected Complexity)。为了分析算法的期望复杂性，往往需要对输入的分布作某种假定。在大多数情况下，这并非很容易，所以我们感兴趣的是分析在某些输入时，使得算法的处理器数及计算时间最大的情况下算法的复杂性，这时的复杂性称为**最坏情况下的复杂性**(Worst-Case Complexity)。因此，在后面各章节我们讨论的复杂性，不言而喻的都是指最坏情况下的复杂性。

---

①书中出现的 log 均以 2 为底。

在 SIMD 计算模型上的并行算法，我们将研究算法的运行时间和所需的处理器数目以及在最坏情况下，它们与问题规模 n 的关系。

(1) **运行时间** $t(n)$: 运行时间就是在并行计算机上求解给定问题所需的时间。即算法从开始执行到算法执行结束所经过的时间，此时间通常由两部份组成：数据从一个处理器经互连网络到达另一个目标处理器的**选路** (Routing) **时间** t_r，以及在一个处理器内执行的算术和逻辑等运算需要的**运算时间** t_c。通常，两者分别用选路时间步以及计算时间步计算之。

(2) **处理器的数目** $p(n)$: 求解给定问题所需的处理器数目，显然它是问题规模 n 的函数。根据当今并行计算机的实际水平，$p(n) > n$ 是不合适的，特别是 n 相当大时，通常 $p(n)$ 应是 n 的**亚线性函数** (Sublinear Function)。虽然 $p(n) = \log n$ 或 $p(n) = n^{1/2}$ 也是 n 的亚线性函数，可它们的取值对于实际的并行计算机而言并不灵活。因此，$p(n)$ 具体选取应视具体并行系统而定。通常将 $p(n)$ 限制为 $p(n) = n^{1-\varepsilon}$，其中 $0 < \varepsilon < 1$。

在 MIMD 计算模型上，对基于共享存贮的算法其时间还应包括同步等。它们同 SIMD 模型上复杂性度量基本一致。而对基于异步通信的分布式计算模型，其算法的衡量标准主要有两个:

(1) **通信复杂性**: 是指算法在整个执行期间所传送的消息总数目；这又分为基本长度消息总数，或传送消息的二进制位数总和;

(2) **时间复杂性**: 是指算法从第一台处理机上开始执行到最后一台处理机上终止的这段时间间隔。

由于在基于异步通信的分布式计算模型中，处理机之间传递的消息到达目的地的时间是有限但不确定的，同时算法的执行时间与处理机互连的拓扑结构紧密相关，要想精确地分析算法的时间复杂性就变得非常困难。因此，目前，估算出的算法复杂性都是假定邻接处理机之间的通信可在 $O(1)$ 时间内完成这一假定基础上得出的。注意：在这种计算模型中一般都假定处理机数目为 $O(n)$，有时假定为 p ($p \leqslant n$)。

2.3.3 并行算法的性能评价

对一个给定的问题，可以设计出许多求解它的并行算法，那么在这众多的并行算法中，孰优孰劣呢？这就需要一个或多个衡量标准。本节将介绍评价一个并行算法的几个公认的衡量标准。

1. 并行算法的成本: $c(n)$

并行算法的成本 $c(n)$，定义为并行算法的运行时间 $t(n)$与并行算法所需的处理器数目 $p(n)$ 的乘积，即

$$c(n) = t(n) \cdot p(n)$$

换句话说，成本等于在最坏情况下求解一问题时所需的执行步数。如果求解一问题的并行算法成本，在数量级上等于最坏情况下串行求解此问题所需的执行步数，则称这样的并行算法是**成本最优的**。

2. 加速（又称加速比，速度倍数）: $S_p(n)$

令 $t_s(n)$ 是求解一问题的最快串行算法在最坏情况下的执行时间，$t_p(n)$ 是求解同一问

题的并行算法在最坏情况下的运行时间，则**加速** (Speedup) 定义为:

$$S_p(n) = t_s(n) / t_p(n)$$

可见，$S_p(n)$ 是反映算法并行性对运行时间的改善程度标准。很清楚，$S_p(n)$ 越大，表明并行算法越好。在理想情况下，$S_p(n) = p(n)$。若一个并行算法，其加速 $S_p(n) = p(n)$，则称这个并行算法是**最优的并行算法** (Optimal Parallel Algorithm)。但实际上要达到 $S_p(n) = p(n)$ 几乎是不可能的。原因在于在绝大多数情况下，所求解的原始问题是不可能分解成 $p(n)$ 个子任务，以及每一个子任务在一个处理器上所需时间为单机时间的 $1/p(n)$，此外，并行计算机为了求解原始任务，对各处理器之间的合作还需要额外开销。事实上，因为任何并行算法都能在一台串行机上模拟实现，因此，$t_s(n) \leqslant p(n) \cdot t_p(n)$，从而

$$1 \leqslant S_p(n) \leqslant p(n)$$

3. **并行算法的效率**: $E_p(n)$

有时候，一个并行算法虽然有好的加速，但处理器的利用率可能很低，特别是当处理器的数目 $p(n)$ 不固定时，$S_p(n)$ 不是一个最好的评价标准，为此引入算法效率的概念。所谓**并行算法的效率**，可定义为算法的加速同处理器数目之比，即

$$E_p(n) = S_p(n) / p(n)$$

可见，$E_p(n)$ 可度量并行系统中处理器的利用情况，由上式可知

$$0 < E_p(n) \leqslant 1$$

在基于异步通信的分布式计算模型中，通信复杂性是主要衡量标准，其次才是计算复杂性。

2.4 并行算法的表示及约定

描述一个算法，速常可使用非形式描述和形式化描述两种。所谓**非形式描述**就是用通常的自然语言来表达；所谓**形式化描述**是使用某种程序设计语言将算法书写出来。在可能的情况下，我们尽量使用算法的形式化描述。选用语言时，其基本要求用该语言所写出的语句不能有二义性，同时要强调直观，易理解，而不苛求严格的语法格式。象串行算法中选用语言一样，类 Algol 或 Pidgin – Algol 或类 Pascal 等语言均可被选用。本书使用的是两者混和体。

在并行算法的表达中，所有串行算法的语句及过程均可被调用，而为了表达并行性，我们引入几条并行语句:

(1) **for** each $i_1, i_2, \cdots, i_k : P$ **pardo**

⋮

endfor

表示编号为 $(i_1,i_2,\cdots,i_k)$ 的处理器，若满足谓词 P 为真，则执行循环体的指令序列，其中 i_k 是一个大于等于 0 的有限整数。

(2)　　upon receiving M message from *u* **do**:

⋮

表示某结点一旦收到一个来自结点 u 的消息 M 后，就执行相应的计算。

注意：为了程序的简洁，在意义明确的前提下，参数类型说明总是省去的，而且“**begin**” “**end**”总出现在一个算法或一个子过程的头部和尾部。我们用 ″←″ 表示赋值，而“⇐”表示在互连网络中，将数据从一个处理器放入另一个处理器的过程。

最后，应强调的是：本书中的各种并行算法的表示，不要求统一格式，视描述方便，或采用非形式化描述，或用形式化描述，且在意义表达明确的前提下，允许并行算法的多种表达方式。

为今后叙述方便，约定一个**有向（无向）图**是 $G(V,E)$，$|V|=n$，$|E|=m$。图的直径用 d 表示。若 G 是有向图，则有向边 $e\in E$，用一对“尖括号”表示，即 $e=\langle u,v\rangle$，这表示 e 的方向由 u 指向 v；若 G 是无向图，我们用一对园括号表示边，如 $e\in E$，$e=(u,v)$。

2.5　并行算法的设计技术

2.5.1　几种基本的设计技术

为解决某一给定问题而设计一个并行算法，基本上存在**两种设计方法**：一种是对这一问题的串行算法进行改造、移植，开发其固有的并行性，使之适用于在并行计算机上运行，这种直接从串行算法转换到并行算法是很有意义的．因为这样一来，大量的串行算法都可移植到并行机上运行．然而，也存在一些限制，盲目地将一个串行算法转换成一个并行算法，常常会导致失败．原因在于某些有效的串行算法固有的顺序性（如 DFS 算法），很难开发出其中的并行性来，而且一个高效率的串行算法也不一定会导致一个高效率的并行算法．相反，有时一个性能不好的串行算法，却有可能直接转换成性能良好、快速的并行算法．另一种是从问题描述本身出发，抛弃现在的此问题串行算法思想，从头开始设计一个全新的并行算法．此种方法看来似乎存在一些技巧，但也不是无章可循的．近年来，人们力图寻求设计并行算法的一些通用策略．但就目前的研究状况来看，这些通用设计技术还十分有限．普遍用到的有：**分而治之策略**[2,3]，以及它的一个变体**瀑布分而治之策略**(Cascading Divide–And–Conquer Strategy)[4]，**路径折叠技术**(Path Doubling Technique)[5]，**树压缩技术**(Tree Cotraction Technique) [6]以及**迭代改进**(Iterated Improvement)**技术**[7]等。下面我们将介绍几种基本设计技术．

1.　分而治之技术

将一个大的问题分解成 p 个大致相等的子问题，然后在每个处理器对相应的子问题求解，最后将各处理器上得到的部分解合并起来，就得到原问题的解。

例如：假定有n个元素a_1，a_2，…，a_n，用p个处理器计算$\sum_{i=1}^{n} a_i$，$p \leqslant n$。

用分而治之技术解决这个问题如下：首先对每个处理器分配元素：每个处理器至多含$\lceil n/p \rceil$个元素；我们分配编号为i的处理器的元素是$a_{(i-1)\lceil n/p \rceil+1}$，…，$a_{i\lceil n/p \rceil}$，$1 \leqslant i \leqslant p-1$，第$p$个处理器分配剩下的$n-\lceil n/p \rceil(p-1)$个元素，即元素$a_{(p-1)\lceil n/p \rceil+1}$，…，$a_n$。然后每个编号为i的处理器计算它内部元素的和$T_i = \sum_{j=(i-1)\lceil n/p \rceil}^{\lceil i \cdot n/p \rceil} a_j$；最后这$p$个处理器合作将这些部分和加起来得出原问题的解。

下面我们分析上述算法复杂性；每个处理器计算部分和需$O(\lceil n/p \rceil)$时间；然后p个处理器计算$\sum_{i=1}^{p} T_i = O(\log p)$时间，故整个算法需$O(n/p + \log p)$时间、$O(p)$处理器。

最近Atallah等人基于Cole的归并排序过程发展了一种他们称之为瀑布分而治之技术，他们论证了这种技术在设计并行算法中，尤其是设计计算几何问题的并行算法中非常有用。

2. 路径折叠技术

Wyllie的路径折叠技术最先应用于链表结构，这里我们通过一个例子来说明这种技术。

已知n个元素的单向链表，计算该链表每个元素后面链接的元素个数。这就是所谓的**表计数问题**(List Ranking Problem)。

设表长为n，除表尾一个元素外，其它每个元素i都有一个指向其后继元素的指针$D(i)$。若表最后一个元素为t，则约定$D(t)$='NIL'，$D(D(t))=D(t)$。设第i个在链表中元素后面链接的元素个数为$C(i)$，则下述过程将完成链表计数。

算法2.1 LIST RANKING PROBLEM

```
procedure LRP(D,C);
  begin
    (1)  for each i: 1≤i≤n  pardo   /*赋初值*/
            C(i)←1;
            if i=t then C(i)←0 endif
         endfor;
    (2)  for k←1 to ⌈logn⌉ do
              for each i: 1≤i≤n pardo
      (2.1)     C(i)←C(i)+C(D(i));   /*同折叠后的邻接顶点值相加*/
      (2.2)     D(i)←D(D(i));
      (2.3)     if i=t then C(i)←0 endif;
              endfor
         endfor
  end.
```

所谓路径折叠，主要体现在过程的第(2.2)步。这个算法的一个形象说明如图 2.2 所示。

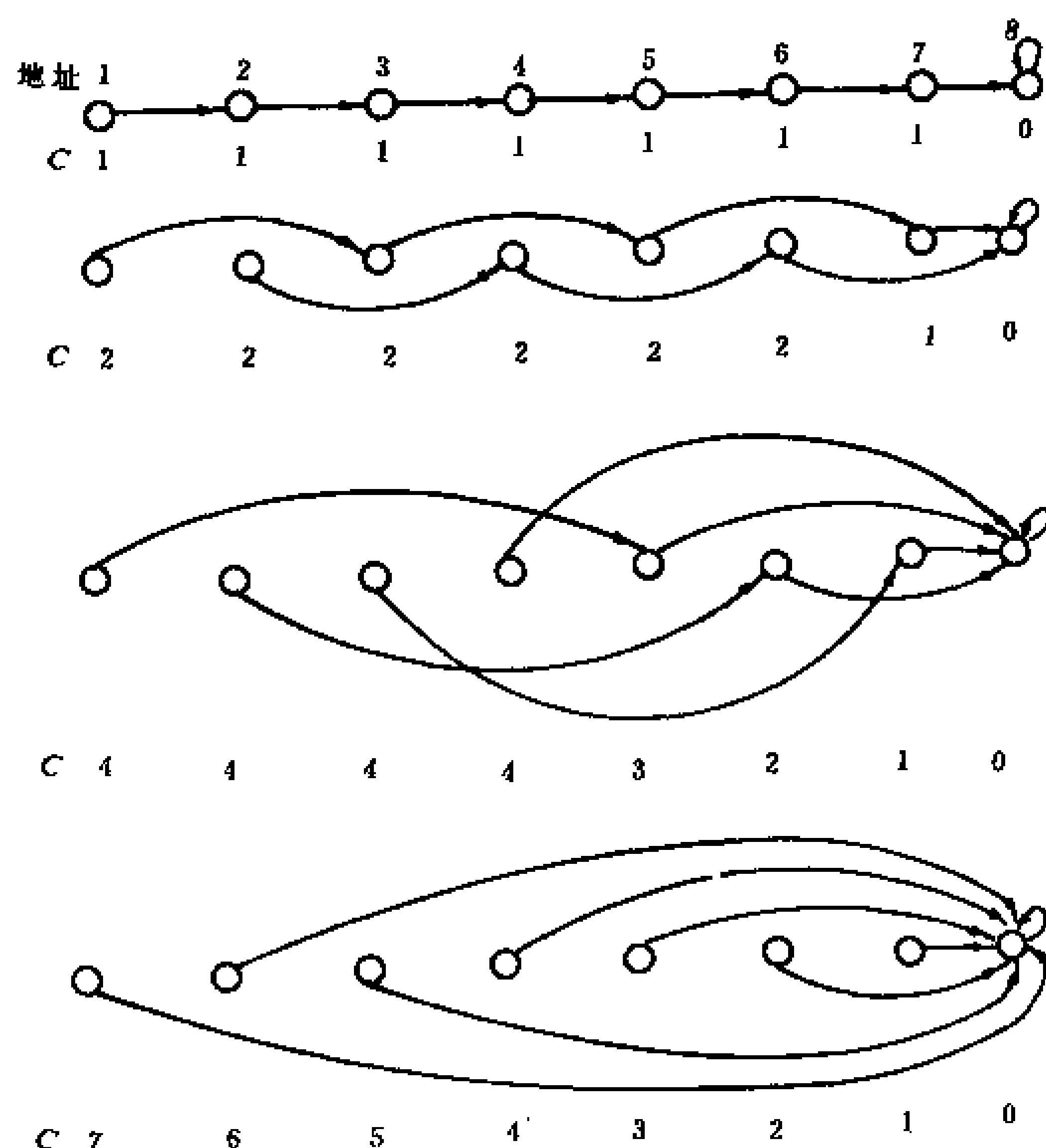


图 2.2　$n=8$ 的链表计数算法执行过程

在 SIMD－CREW PRAM 上，上述过程需 $O(\log n)$ 时间、$O(n)$ 个处理器。

这种路径折叠技术已广泛地应用到并行算法设计中。

3. **迭代改进技术**

迭代改进技术最初是由 Karp 等人提出的。这种技术并不象其它技术那样，一开始就急于寻求问题的解，而是通过许多次迭代逐步改进，最后才给出问题的解。每次迭代称为一个阶段，每个阶段的工作是使后选解的个数减少一个常量因子，这样，一个规模 n 的问题需 $O(\log n)$ 次迭代就可得到原问题的解。

下面通过一个例子来说明这种技术的应用。首先我们定义图 $G(V,E)$ 的**独立集**。一个图 G 的一个独立集 I 是顶点集 V 的子集 $I\subseteq V$，且 I 中任意两个顶点都没有边连接。图 G 的一个**极大独立集** (Maximal Indepent Set) I 指的是：I 是一个独立集，且它不真正包含在另一个独立集中。今后我们将极大独立集简记为 MIS。

令 $S\subseteq V$，$N_G(S)$ 定义为 S 在 G 中的邻集，即 $N_G(S)=\{w\mid w\in V$ 且存在 $u\in S$，$(u,w)\in E\}$。有了上述定义，下面举一例子。

已知一无向图 $G(V,E)$，试求 G 的 MIS。Karp 等人曾用迭代改进技术对这一问题建议了一个并行算法，他们的算法简单地描述如下：

算法2.2　MAXIMAL INDEPENDENT SET PROBLEM

procedure Maximal_Independent_Set (G,V,E);

```
begin
  (1) I←φ; H←V;    /* I是G的极大独立集 */
  (2) while H≠φ do
  (2.1) S←an independent set in induced subgraph H;
  (2.2) I←I∪S;
  (2.3) H←H−(S∪N_H(S));
        /* 除去独立集S及其邻接顶点后，剩下的顶点导出的子图 */
      endwhile
end.
```

上述算法始终维持一个集合 I，最终它将成为极大独立集。在每一迭代阶段(即 while 循环执行一次)，不在 I 内的顶点数目、或同 I 内顶点相邻的顶点数目都显著地减少。

Karp 等人证明了在 SIMD − CREW PRAM 上，极大独立集问题的并行算法需 $O(\log^4 n)$ 时间、$O(n^3/\log^3 n)$ 处理器。

运用迭代改进技术，已经研究出了许多有效的 MIS 并行算法及平面图着色的并行算法[8,9,10]。

树压缩技术是基于树结构上执行一系列操作技术，这里就不介绍了。另外近年来又发展了所谓的**确定性硬币投掷**(Deterministic Coin Tossing)**技术**[11] 以及**破对称**(Symmetry − Breaking)**技术**[12]，这些技术都是设计有效并行算法的有用技术。

2.5.2 设计并行算法应注意的几个问题

在设计一个并行算法时，有一些基本的因素应牢记在心中。

(1) **计算问题的并行性开发**：对于一个具有内在并行性的计算问题，我们必须根据不同的计算模型，改变计算问题的运算操作之间的相互关系，分析其相关性，将其划分成若干个能彼此并行执行的子问题。注意，使用不同的划分方式，将会产生不同风格的并行算法。

(2) **必须考虑通信成本**：在基于共享存贮的并行计算模型上，由于处理器之间的通信是通过全局共享存贮变量完成的，因而通信开销常常忽略不计。但在基于互连网络结构的并行计算模型上，在决定并行算法的复杂性时，不考虑通信开销是不行的。有时候通信复杂性比计算复杂性还要高，也就是说，数据在处理器之间选路所花费的时间比实际处理、变换所花费的时间还要长。例如，在某一特定的多计算机上执行一次浮点加法运算需 t 个单位时间，而将一个浮点数从一个处理器传送到另一个处理器需要 $100t$ 个单位时间。若在 p 个处理器的并行系统中对 n 个浮点数进行加法运算，且假定这 n 个浮点数开始时都在局部存贮器中，那么，当 $n \leqslant 101$ 时，不管处理器的个数 p 多大，在由单个处理器的计算机上执行这 n 个数的浮点加法总比用多个处理器同时并行执行浮点加要快。

(3) **算法有赖于计算模型**：一个算法的性能在不同计算模型上差别很大。有时候，

是由于如上所述的通信开销所致；但有时候是由另外一些因素引起的。例如，在 SIMD 计算模型上同步是自动实现的；但在 MIMD 计算模型上它要通过很费时间的软件来实现。所以，通常在同步之间执行很少运算的一类算法，在 MIMD 共享存贮机器上运行的效率是很差的。

(4) **算法与数据的存贮分布有关**：例如，在作矩阵运算时，矩阵元素在各存贮模块中可以有不同的存贮方式。假定有 4 个存贮模块，矩阵元素按如下方式存贮：

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \quad \text{存贮模块编号}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

如果按行或对角线方式存取矩阵元素，则 4 个存贮模块可以同时工作；而如果按列存取矩阵元素，则 4 个模块不能同时工作。

(5) **并行算法与串行算法的性质不同**：由于并行算法的计算方式同串行算法不同，同时并行算法还要考虑通信开销以及辅助开销，而不具有顺序算法严格的顺序继承性。所以在诸如收敛性、稳定性和舍入误差等方面，并行算法的性质都不同于串行算法。对这些问题，在设计并行算法时都要给予分析和验证。

2.6 小　　结

本章主要阐述了并行算法的基本概念、衡量标准及性能评价标准。并行算法的衡量参数主要有运行时间、使用的处理器数目等。分布式算法还包括通信复杂性。并行算法性能评价的主要指标是：成本、加速、效率等。

本章除了介绍了并行算法的基本概念外，还介绍了并行算法的几种常见的设计技术：如分而治之技术、路径折叠技术和迭代改进技术。同时约定了并行算法的表示。最后，指出了在设计并行算法时应注意的若干问题。

参 考 文 献

[1] Kung H T. *The Structure of Parallel Algorithms*. In Advances in Computer. 19, Academic Press, NY, 1980, 65–112

[2] Horowitz E, Zorat A. Divide–and–Conqure for Parallel Processing, *IEEE Trans. Computer*, C–32 (1983), 582–585

[3] 唐策善，梁维发，分治策略设计并行算法．微电子学与计算机，7 (4)，1990，17–20

[4] Atallah M J, Cole R, Goodrich M T. Cascading Divide–and–Conquer: A Technique for Desiging Parllel Algorithms. *SIAM J. Comput.*, 18(3), 1989, 499–532

[5] Wyllie J C. The Complexity of Computation, Technical Report TR79–387, Dept. of Comput. Sci.

Cornell Univ., Ithaca, NY, 1979

[6] Miller G L, Reif J H. Parallel Tree Contraction and Its Application. 26th *Annu. Sympo. On Foundations of Computer Science, IEEE*, 1985, 478–489

[7] Karp R M, Wigderson A. A Fast Parallel Algorithm for the Maximal Independent Set Problem . *Proc. 16th ACM Sympo. On Theory of Computing*, 1984, 266–272

[8] Hagerup T. Chrobak M, Diks K. Optimal Parallel 5–Coloring of Plenear Graphs. *SIAM J. Comput.*, 18(2), 1989, 288–300

[9] Naor J. A Fast Parallel Coloring of Planar Graphs with Five Colors, *Inform. Proc. Lett.* 25, 1987, 51–53

[10] Goldberg M, Spencer T. A New Parallel Algorithm for the Maximal Independent Set Problem, *SIAM J. Comput.*, 18(2), 1989, 419–427

[11] Cole R, Vishkin U. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Inform. and Control*, 70, 1986, 32–53

[12] Goldberg A V, Plotkin S A, Shannon G E. Parallel Symmetry–Breaking in Sparse Graphs. *in Proc. 19th Annu. ACM Sympo. On Theory of Computing*, 1987, 315–324

第三章 图 的 搜 索

3.1 图的并行搜索

搜索是解决图论问题的一种基本技术，它是许多图论算法的基础．传统的搜索技术有深度优先搜索（DFS）及宽度优先搜索（BFS）两种搜索技术。这里介绍这两种技术以及它们结合在一起的宽–深优先搜索技术的并行化．

假定计算模型是 SIMD–CREW PRAM． Reghbati 和 Corneil 曾给出了三种图的并行搜索方法[1]．我们将介绍这些方法．

3.1.1 算法的基本原理

在计算机中，图 $G(V,E)$都用适当的数据结构表示，如邻接矩阵、邻接表等．由于用邻接矩阵找边的时间开销较大，这里采用邻接表来表示图．在并行环境下，一个图怎样才能很快地被搜索呢？

开始时，令主表为空．从图中任选一个顶点作为开始搜索的顶点，并将其放人主表中．在任何一次搜索过程中，每个处理器首先把自己的子表置空，然后从主表中选择一个待搜索的的顶点，每个处理器各自检查这个顶点的一条或几条关联边．若关联边连接一个未被搜索的顶点，则把它放人该处理器的子表中．各子表中存放着将要并人主表的顶点。在某个间隔点，把各处理器产生的子表链接到一起并人主表中。

假定消耗时间的操作有顶点选择、表链接和结合过程，并假定选择一个顶点需要这类操作中的一种。

对顺序算法而言，仅存在一个主表，每检查一条关联边至多只有一个未搜索顶点加人主表中．设 d_i 为顶点 $i\in V$ 的度数，那末搜索一个图的顺序算法时间上界为

$$T_1 = \sum_{i=1}^{n}(d_i + 1) = 2m + n$$

现在让我们讨论在有 p 个处理器的计算模型上，并行搜索算法及其时间复杂性 $(1 < p\leqslant n)$.

3.1.2 $p-$ 深度优先搜索

在 $p-$ 深度优先搜索算法中，主表是一个后进先出（LIFO）表，即是栈．一旦从主表选择了一个待搜索的顶点 i．则与它相连的至多 p 条边将同时进行检查，每个处理器将所发现的未被搜索过的顶点放人自己的子表中，然后把这些子表链接起来并人主表中；下一次选择的待搜索顶点是本次搜索到的一个顶点，若待搜索顶点所有关联边均被检查过，则从主表中删除这个顶点，当主表为空时，搜索过程终止．

定理 3.1 在 SIMD – CREW PRAM 上，对一个图 $G(V,E)$ 执行 $p-$ 深度优先搜索的时

间 T_p^1 满足:

$$T_p^1 \leqslant T_1(\lceil \log p \rceil + 1)/p + n(\lceil \log p \rceil + 1)$$

证明 每次搜索时，所发现的未被搜索过的顶点总是位于子表中。将 p 个处理器的子表链接成一个表需 $\lceil \log p \rceil + 1$ 时间，而检查一个待搜索顶点 $i \in V$ 的所有边，需 $\lceil d_i / p \rceil + 1$ 次搜索，故 $p-$ 深度优先搜索的时间 T_p^1 为

$$\begin{aligned} T_p^1 &= \sum_{i=1}^{n}(\lceil d_i / p \rceil + 1)(\lceil \log p \rceil + 1) \\ &= \sum_{i=1}^{n}\lceil d_i / p \rceil (\lceil \log p \rceil + 1) + n(\lceil \log p \rceil + 1) \\ &\leqslant T_1(\lceil \log p \rceil + 1)/p + n(\lceil \log p \rceil + 1) \end{aligned}$$

3.1.3 $p-$ 宽深优先搜索

在并行宽深优先搜索过程中，从主表中选择一个待搜索顶点 i，并从主表中删除这个顶点。每个处理器至多检查这个顶点的 $\lceil d_i / p \rceil + 1$ 条关联边，同时把发现未被搜索过的顶点加入自己的子表中，然后将子表链接起来加入主表中，下一次搜索时，选择本次搜索到的一个未被搜索过的顶点作为待搜索顶点，否则从主表中选择最后加入的顶点作为待搜索顶点，直到主表为空时，搜索过程终止。

定理 3.2 在 SIMD－CREW PRAM 上，对一个图 $G(V,E)$ 执行 $p-$ 宽深优先搜索的时间 T_p^2 满足:

$$T_p^2 \leqslant T_1 / p + n(\lceil \log p \rceil + 3)$$

证明 每次搜索时，检查待搜索顶点 $i \in V$ 的关联边需 $\lceil d_i / p \rceil + 1$ 时间，然后将每个处理器子表链接起来结合到主表中需 $\lceil d_i / p \rceil + 1$ 时间，故 $p-$ 宽深优先搜索的时间 T_p^2 为

$$\begin{aligned} T_p^2 &= \sum_{i=1}^{n}(\lceil d_i / p \rceil + 1 + \lceil \log p \rceil + 1) \\ &\leqslant \sum_{i=1}^{n}(\lceil d_i / p \rceil + \lceil \log p \rceil + 3) \\ &\leqslant T_1 / p + n(\lceil \log p \rceil + 3) \end{aligned}$$

3.1.4 $p-$ 宽度优先搜索

在 $p-$ 宽度优先搜索过程中，主表是一个先进先出（FIFO）的队列．一旦从主表中选择了一个待搜索顶点 i，则把 i 从主表中删除，然后 p 个处理器检查 i 的一些关联边，并把发现的未被搜索过的顶点放入子表中；下一次搜索仍从主表中选择待搜索顶点，直到主表空时，才把子表链接起来放入主表中。当所有子表链接后仍为空时，则搜索过程结束。

同前面两种并行搜索技术相比，并行宽度优先需要更少链接结合步。因为处理器在进行搜索树的第 $(l+1)$ 层之前，已检查过（搜索过）前 l 层的所有顶点，这样，搜索树的每层仅需一个结合步 $(1 \leqslant l \leqslant n-1)$。

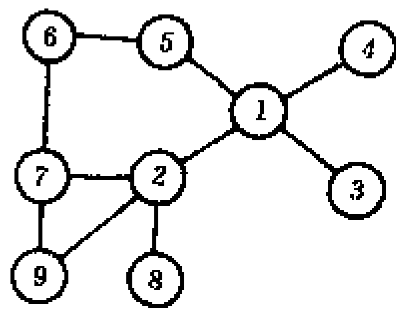
定理 3.3 在 SIMD－CREW PRAM 上，对一个图 $G(V,E)$ 执行 p— 宽度优先搜索的时间 T_p^3 满足：

$$T_p^3 \leqslant T_1 / p + L \lceil \log p \rceil + 2n$$

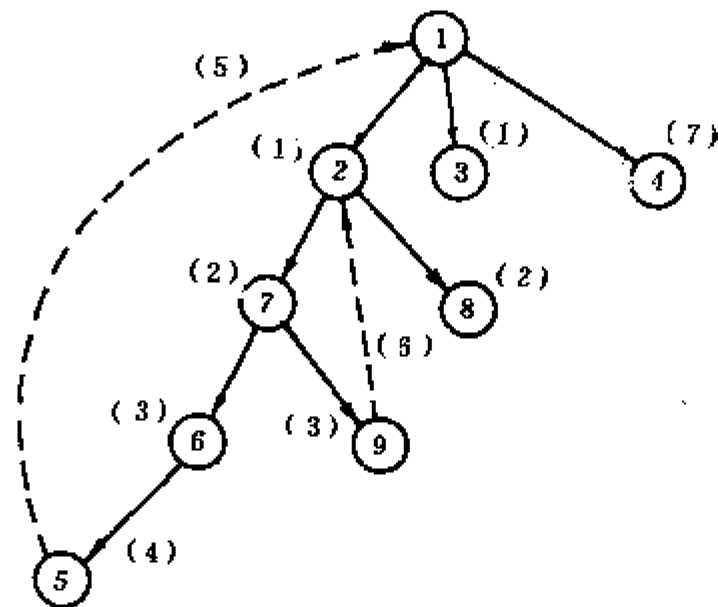
这里 L 是指从开始顶点到最远顶点的距离。

证明 因为搜索图的每层以后，链接结合步需要 $\lceil \log p \rceil + 1$ 时间，每个顶点 i 的关联边检查需要 $\lceil d_i / p \rceil + 1$ 时间，整个图从始点开始搜索有 L 层，则 p— 宽度优先搜索的时间 T_p^3 为：

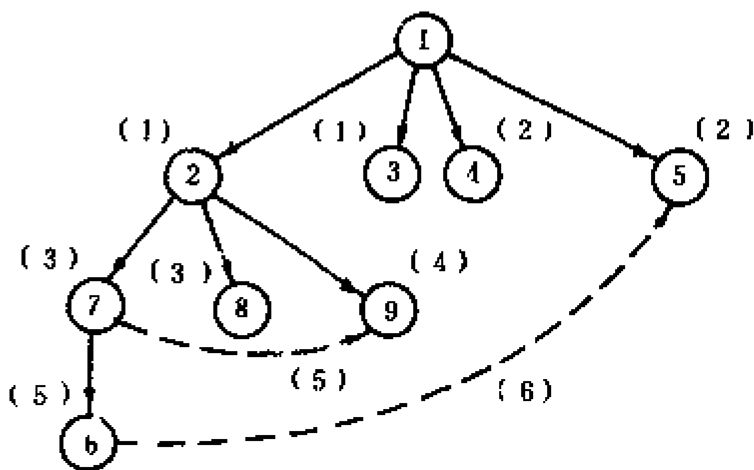
$$\begin{aligned} T_p^3 &= \sum_{i=1}^{n} (\lceil d_i / p \rceil + 1) + L(\lceil \log p \rceil + 1) \\ &\leqslant \sum_{i=1}^{n} \lceil d_i / p \rceil + L \cdot \lceil \log p \rceil + 2n \\ &\leqslant T_1 / p + L \cdot \lceil \log p \rceil + 2n \end{aligned}$$



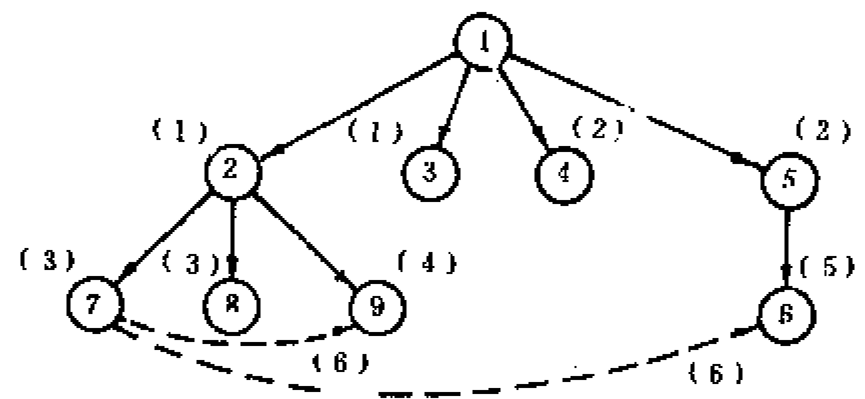
(a) 图 G



(b) 两个处理器的 p—深度优先搜索



(c) 两个处理器的 p—宽深优先搜索



(d) 两个处理器的 p—宽度优先搜索

图 3.1 图的并行搜索

图 3.1 示出了上述三种并行搜索过程，其中括号里的数字表示遍历的次序，虚线表示所搜索到的顶点已被搜索过。

3.2 图的分布式搜索

在异步通信的通信网络上，研究图的搜索技术具有重要意义。因为网络通信的异步性，每个网络结点只知道与它邻接的结点这一局部信息，要了解网络全貌，就必须通过向网络发送消息，对网络进行搜索，这样，通过应答消息才能知道网络的情况，如网络中的结点数目、网络的连通性等。本节将介绍纯遍历搜索算法、分布式深度优先搜索算法以及分布式宽度优先搜索算法。

3.2.1 纯遍历搜索(Pure Traversal Searching)算法

1. 算法的基本原理[8]

首先，起始结点 $s\in V$ 进行初始化工作，然后 s 向它的邻接结点发送访问消息 VISIT。当一个结点 $v\in V$ 首次收到 VISIT 消息，把发送者当作自己的父结点 $F(v)$；若一个已访问过的结点收到 VISIT 消息，就给发送者一个 ACK 应答消息。同时，若 v 除连接父结点外还有其它的关联边，则向这些关联边发送 VISIT 消息。若 v 没有其它的关联边，则向父结点送回 RETURN 应答消息。当一个结点收到所有的应答且不是根结点（起始点）时，则向父结点送回 RETURN 消息。当根结点收到所有应答消息后，向所有儿子结点发送 TERM 消息。终止算法的执行。一个结点收到 TERM 消息，若非叶结点，则向儿子结点传送 TERM 消息，终止算法执行；若是叶结点，则算法终止。

2. 算法的形式化描述

算法 3.1 PURE TRAVERSAL SEARCHING

```
输入：每个结点 v 的邻居集 N(v)={u|(u,v)∈E}；
输出：以起始点 s 为根的分布式生成树，每个结点有父亲F(s)及儿子集合SONS(v).
begin
根结点 s 的算法：
初始化：
F(s)←s；   SONS(s)←φ；
for each v∈N(s) do   /*起动算法执行*/
  send VISIT message to v
endfor；
upon receiving VISIT message from v do：
  send ACK message to v；   /*对消息发送者的应答*/
upon receiving RETURN message or ACK message from all v∈N(s) do：
  send TERM message to v；
  stop itself algorithmic execution；
upon receiving RETURN message from v do：
  SONS(s)←SONS(s)∪{v}；   /*搜索树结点 s 的儿子集合*/
```

一般结点$v(\neq s)$的算法:

```
初始化:
F(v)←0;    SONS(v)←φ;
upon receiving VISIT message from u do:
  if F(v)=0    /*第一次收到VISIT信息*/
    then  F(v)←u;
          N(v)←N(v)−{u};
          if N(v)≠φ  then send VISIT message to all w∈N(v)
                          else send RETURN message to F(v)
          endif
    else send ACK message to u
  endif;
upon receiving RETURN message from u do:
  SONS(v)←SONS(v)∪{u};
  if receiving RETURN message or ACK message from all w∈N(v)
   then send RETURN to F(v)
  endif;
upon receiving ACK message from u do:
  if receiving RETURN or ACK message from all w∈N(v)
    then send RETURN message to F(v)
  endif;
upon receiving TERM message from u do:
  if SONS(v)≠φ
       then send TERM message to all w∈SONS(v)
  endif;
  stop executing algorithm
end.
```

定理 3.4 在异步通信的分布式计算模型上，无向连通图 $G(V,E)$，$|V|=n$，$|E|=m$ 的纯遍历搜索算法 3.1 需要的通信复杂性为 $O(m)$，时间复杂性为 $O(n)$.

证明 由于每条边至多有两个方向相反的 VISIT 消息以及这两个消息对应的应答 RETURN 消息或 ACK 消息，故通信复杂性为 $4m=O(m)$、设从起始点到最远距离的结点间长度为 L，则时间复杂性为 $O(L)=O(n)$.

3.2.2 深度优先搜索算法

1. 算法的基本原理

Cheung 曾建议了第一个分布式深度优先搜索算法．算法的基本思想是：假定 u 是最新标号的结点，立即搜索与 u 相邻的一个未标号的结点 v，然后 v 被标号，搜索将从 v 开

始进行，若不存在未标号结点 v，则搜索返回到 w（这里 u 是从 w 搜索时得到的）。

2. **算法的形式化描述**

算法3.2 DISTRIBUTED DFS

输入：每个结点 $v\in V$ 的邻居集 $N(v)=\{u|(u,v)\in E\}$；

输出：根为起始点的 DFS 树，每个结点 v 有父亲 $F(v)$，$v\in V$。

```
begin
  根结点 s 的算法:
  初始化:
  F(s) ← s ;
  E(s) ← N(s) ;
  send ( label ; s ) message to v, for one v∈E(s) ;
  一般结点的算法 ( 位于结点 r ) :
  upon receiving ( label ; d ) from dispatcher d do :
    if r has not been labeled
       then F(r) ← d ;   label r as F(r) ;  E(r) ← N(r) − {d} ;
            if E(r) ≠ Φ
              then send ( label ; r ) to v, for one v∈E(r)
              else send ( echo ; r ) to F(r)
            endif
       else send ( echo ; r ) to d
    endif ;
  upon receiving ( echo ; d ) from dispatcher d do :
    E(r) ← E(r) − {d} ;
    if E(r) ≠ Φ
      then send ( label ; r ) to v, for one v∈E(r)
      else if r ≠ s
              then send ( echo ; r ) to F(r)
              else  the algorithm terminates
           endif
    endif
  end .
```

3. **算法的正确性证明**

为了证明分布式算法实际上做的同它应该完成的功能一样，是一项困难的工作。下面我们非形式化证明算法 3.2 的正确性。

引理 3.1 从起始点 s 开始，图的每个顶点都被标号且仅标号一次。

证明 因为结点 s 首先被标号，然后 s 向 $N(s)$ 中所有顶点发标号消息，又因为图是连通

的，故集合$\bigcup_{r\in V}E(r)$内所有结点最终都收到标号消息，因而它们都被标号，同时每个结点仅当未标号时才被标号，所以每个结点只标号了一次。

引理3.2 算法 3.2 最终会终止。

证明 显然每个标号消息最终将由应答消息响应，因此，最终 $E(s)$ 将变成空集合，算法将在结点 s 处终止，而且也仅在此结点终止。

定理 3.5 分布式深度优先搜索算法 3.2 能正确地完成搜索。

证明 我们将证明深度优先搜索的两个独特性质。1) 每个结点 r 的邻接点是一个接着一个的标号后被搜索的。在算法中，每次 r 发布标号消息仅给一个邻接点，r 发布标号消息给另一个邻接点仅当收到第一个标号消息的应答后。2) 搜索过程开始应用于 $F(r)$ 的另一个儿子仅当目前的儿子 r 以及它的所有子孙都已搜索过。算法也正是这样做的。

4. 算法的复杂性分析

定理 3.6 在异步通信的分布式计算模型上，对图 $G(V,E)$，$|V|=n$，$|E|=m$ 执行深度优先搜索的算法 3.2 需要的通信复杂性和时间复杂性均为 $O(m)$。

证明 算法的通信复杂性是指在算法执行期间标号消息数目与应答消息数目之和。在每个结点 r，应答集合 $E(r)$ 可作这两个数的计数器，因此，算法通信复杂性为$\sum_{r\in r}|E(r)|=O(m)$，又因遍历是顺序的，故时间复杂性为 $O(m)$。

算法 3.2 的时间复杂性较高，这主要是对所有边顺序遍历的缘故。Awerbuch 改进了这一个算法，使得在通信复杂性不变的情况下，时间复杂性降至 $O(n)$[9]。下面我们介绍这一改进算法。

3.2.3 改进的深度优先搜索算法

1. 基本原理

1985 年 Awerbuch 对算法 3.2 作了改进，改进后的算法基本上与算法 3.2 一样，不同的是对回边（从搜索结点到一个已标号结点之间的边）的处理方法有所差别，即对回边执行并行遍历。这样，只在树边上进行顺序遍历，使得时间复杂性减到 $O(n)$。

算法执行如下：当一个结点收到 DISCOVER 消息，那么这个结点是第一次被访问，它成为搜索的活动中心，消息的发送者就成了它的父结点。这个结点发送 VISITED 消息到除父结点外的所有其它邻接结点。若除父结点外没有其它邻接结点，则向父结点送回 RETURN 应答消息。当一个结点收到 VISITED 消息，它知道消息的发送者已被访问过，从自己邻接结点集合中删除发送者，表示不再访问发送者，然后送回一个 ACK 消息的应答。当发送 VISITED 消息的结点收到所有邻接结点的应答后，活动中心发送一个 RETURN 消息给自己。当一个结点收到 RETURN 消息，那么搜索将从此结点开始。也就是说，若存在一个未访问的邻接结点，DISCOVER 消息将发送给它，这个邻接结点将标记访问过；否则，RETURN 消息将返回到父结点。

2. 算法的形式化描述

算法3.3 IMPROVED DISTRIBUTED DFS

算法中的消息类型：

DISCOVER：到达一个未被访问过的结点的消息；

RETURN：将活动中心返回到一个已访问过的结点的消息；

VISITED：第一次被访问的结点发送的消息；

ACK：对VISITED消息的响应，一种应答消息。

结点 i 含有的变量有

Neighbour(i)：结点 i 的邻接结点集合，也是算法的输入；

Father(i)：在 DFS 树中，结点 i 的父结点，也是算法的输出，根 s 的 Father(s) = s；

Unvisited(i)：Neighbour(i)的子集合，它由尚未收到VISITED信息的邻接结点组成。初始化时，Unvisited(i) = Neighbour(i)；

flag (v,j)：一个二元标志，在 i 向 j 送出VISITED之后，j 送回ACK应答之前，flag(i,j) = 1，初值 flag(i,j) = 0；

```
begin
   根结点 s 的算法:
     初始化:
   Father(s) ← s ;
   Unvisited(s) ← Neighbour(i) ;
   send DISCOVER message to itself;
   一般结点i(≠s)的初始化算法:
   Father(i) ← 0;    Unvisited(i) ← Neighbour(i ) ;
   结点 i 的算法:
   upon receiving DISCOVER message from j  do:
         / * i is visited for the first time * /
     Father(i) ← j ;
     if Neighbour(i) ≠ {j}
       then send VISITED message to q, for all q∈Neighbour(i) − {j}
       else send RETURN message to j   / *  j is the only neighbour of i  * /
     endif;
   upon receiving RETURN message from q do:
       / *  the search is resumed from node i which was already visited in the past * /
     if Unvisited(i) ≠ Φ
       then send DISCOVER message to k, for one k∈Unvisited(i) ;
         Unvisited(i) ← Unvisited(i) − {k}
       else   / * all the neighbour were visited * /
         if Father(i) ≠ i   / * i isn't node of root * /
           then send RETURN message to Father(i)   / * backtracking * /
           else stop, the algorithm has terminated
         endif
     endif;
```

```
    upon receiving VISITED message from k do:
      Unvisited(i) ← Unvisitd(i) − {k};
      send ACK message to k;
    upon receiving ACK message from j do:
      flag(i,j) ← 0;
      if flag(i,q) = 0 for all q∈Neighbour(i) then send RETURN to itself endif
end.
```

3. **算法的复杂性分析**

定理 3.7 在异步通信的分布式计算模型上，算法 3.3 计算图 $G(V,E)$ 的 DFS 树所需的通信复杂性为 $O(m)$，时间复杂性为 $O(n)$。

证明 不难证明，当活动中心到达某个结点时，这个结点准确地知道到目前为止，它的那些邻接结点已访问过。因此，DISCOVER 决不会送到回边上，这也是节省时间的原因。注意在每条树边上，仅有两个方向相反的消息 DISCOVER 及 RETURN 传送，而且，每个结点向它的每个邻接结点(除父结点外)发送 VISITED 消息，每个 VISITED 消息有一个响应消息 ACK。因此，整个算法的通信复杂性为 $4m = O(m)$，另外，只有树边顺序的遍历，故时间复杂性为 $O(n)$。

值得注意的是：分布式深度优先搜索的研究一直受到广泛的重视。1987 年 Lakshmanan 等人进一步改进了算法 3.3，他们的算法的通信复杂性为 $4m-(n-1)$，时间复杂性为 $2n-2$。该算法的时间复杂性是最优的[11]。

3.2.4 宽度优先搜索算法

1. **算法的基本原理**

1983 年 Cheung 建议的分布式宽度优先搜索算法[7] 的基本思想是：假定结点 u 已标号但还未扫描过，为了扫描 u，先给 u 的所有未标号的邻接结点标号，在扫描 u 之后立即扫描标过号的结点。

算法开始于起始结点 $s\in V$，s 发送层号 0 到所有邻接结点。一般来讲，当一个结点 r 收到一个层号 l 消息，它将 l 与自己的层号 l_r（初始值为 ∞）进行比较。若 $l+1 \geqslant l_r$，则忽视 l；否则用 $l+1$ 替换 l_r，然后 r 把层号 $l+1$ 发送到所有邻接结点。在算法终止时，离根 s 的最短路径为 l_r 的结点得到一个层号 l_r。

2. **算法的形式化描述**

算法3.4 DISTRIBUTED BFS

输入：每个结点 $v\in V$ 的邻接结点集合$N(v)$；

输出：一个根为 s 的BFS树，每个树结点 v 有父结点$F(v)$，层号l_v，以及儿子集
SONS(v)。

```
begin
  根结点 s 的初始化算法:
  l_s ← 0;   E(s) ← N(s);   SONS(s) ← N(s);   F(s) ← s;
  send (label; l_s, s) to v, for all v∈E(s);
  一般结点 r(≠s) 的初始化算法:
  l_r ← ∞;   SONS(r) ← φ;   F(r) ← 0;
  在结点 r 处的算法:
  upon receiving (label; l, d) from sender d do:
     if l_r ≠ ∞   /* r is labeled */
       then if l + 1 < l_r
                then send (delete; l_r − 1, r) to v ,for v∈F(r);
                     F(r) ← d;   l_r ← l + 1;   E(r) ← N(r) − {d};
                     if E(r) ≠ φ
                       then SONS(r) ← E(r);
                            send (label; l + 1, r) to v ,for all v∈E(r)
                       else send (echo; l, r) to d
                     endif
                 else send (delete; l, r) to d
           endif
       else F(r) ← d ;  l(r) ← l + 1;  E(r) ← N(r) − {d};
             if E(r) ≠ φ
               then SONS(r) ← E(r);
                    send(label; l + 1, r) to v, for all v∈E(r)
               else send (echo; l, r) to d
             endif
     endif;
   upon receiving (echo; l, d) from sender d do:
    if l ≠ l_r then wait
             else E(r) ← E(r) − {d};
                  if (E(r) = φ) ∧ (r = s) then The algorithm terminates endif;
                  if (E(r) = φ) ∧ (r ≠ s) then send(echo; l_r − 1, r) to F(r)
                  endif;
                  if E(r) ≠ φ then wait endif
    endif;
   upon receiving (delete; l, d) from sender d do:
    if l≠l_r  /* this state takes care of the case that r has now been
          updated with a lower layer number and is no longer the father of d */
```

```
        then wait
        else E(r) ← E(r) − {d};
              SONS(r) ← SONS(r) − {d};
              if E(r) = φ  then send (echo; l_r − 1, r) to F(r)
                                 else wait
              endif
      endif
  end.
```

3. 算法的复杂性分析

对算法 3.4 的复杂性分析比较困难，这是因为算法的通信复杂性依赖于结点所在的层号被修改的次数，而层号修改次数又依赖于网络的拓扑结构以及检查邻接结点的次序。故我们只能给出在最坏情况下算法的复杂性。

定理 3.8 在异步通信的分布式计算模型上，对图 $G(V,E)$，$|V| = n$ 执行宽度优先搜索的算法 3.4 需要的通信复杂性为 $O(n^3)$，时间复杂性为 $O(n)$。

证明 设在这种异步通信网络上对一个含 n 个结点的完全有向图 $G(V,E)$ 进行宽度优先搜索。在最坏情况下，起始点 s 只标号一次，而其它结点共标号了 $\sum_{i=1}^{n-1} i = n(n-1)/2$ 次。一个结点每次标号后都要发送出 $n-2$ 个标号消息（s 除外，它发出 $n-1$ 个标号消息），每个标号消息将由应答消息或删除消息响应。因此整个算法的通信复杂性为

$$2(n-1) + 2(n-2)n(n-1)/2 = O(n^3)$$

假定一个消息在一条通信链上传送时间至多为一个常量 c，那么，在算法开始时，根结点 s 首先得到正确标号；在第 $c+1$ 个时间间隔，将至少有另一个结点得到正确标号；在第 $2c+1$ 个时间间隔，将至少又有一个结点得到正确标号，…，故在第 $nc+1$ 个时间间隔，所有结点都得到正确标号。所以算法的时间复杂性为 $O(n)$。

分布式宽度优先搜索算法 3.4 尽管时间复杂性已达到最优，但通信开销太高。在分布式处理过程中，这两者之间存在着一种动态的平衡。下面我们将介绍一个通信及时间复杂性均为 $O(n^2)$ 的分布式宽度优先搜索算法。

3.2.5 改进的宽度优先搜索算法

Zhu 等人基于分布式系统中通信复杂性与时间复杂性之间存在着某种平衡关系，他们建议了一个通信复杂性和时间复杂性均为 $O(n^2)$ 的 BFS 树算法[10]。

1. 算法的基本原理

算法的主要思想是：BFS 树是一层一层的创建起来的。事实上，算法是由多次循环构成的。每次循环构造了 BFS 树新的一层。一般说来，每次循环又分为两个阶段：第一阶段为标号阶段，第二阶段为应答阶段。在标号阶段，开始从根结点 s 出发，沿树边遍历至树叶结点，叶结点广播 LABEL 消息到那些未访问过的邻接结点，目的在于创建另一个结

点层，这样就结束了第一阶段．在应答阶段：始于这些新的树叶结点，它们送回 ECHO 消息沿树边抵达根结点，这样就终止了这次循环，然后根结点根据接收的消息决定是否还进行下一次循环．

2. 算法的形式化描述

算法3.5 IMPROVED DISTRIBUTED BFS

输入：每个结点 v 的邻接结点集合Neighbour(v)；

输出：一个根在s 的BFS树，其中每个结点v 有父结点Father(v)及层号Level(v)以及儿子 Sons(v)等局部变量．

算法用到的消息类型：

INIT：发送给开始结点、初始化遍历；

LABEL(k, lev)：结点 k 送给请求标号结点 i 的消息，给结点 i 标层号为$lev+1$；

ECHO (k, Status)· 结点k 送出的响应消息，通过k 扩展BFS树的可能性消息由状态参数的三种可能值表达．

(a) 'keepon'：结点i 继续给结点k 发标号消息，因为k 的某个邻接结点尚未标号；

(b) 'stop'：结点 i 停止向结点 k 发标号消息，因为 k 不可能是 i 的一个儿子；

(c) 'end'：结点 k 是扩展的末端，因为它没有未标号的邻接结点．

结点i 的局部变量：

Neighbour(i)：结点 i 的邻接结点集合；

Labeled(i)：逻辑变量，当且仅当 i 被标号后其值为"真"；

Father(i)：在BFS树中结点 i 的父结点，根结点的父结点是自身；

Level(i)：在BFS树中，结点 i 的层号；

Unvisiteds(i)：Neighbour(i)的一个子集，结点 i 应向这些邻接结点发送标号消息；

Sons(i)：在BFS树中结点 i 的儿子集合；

Echoed(j)：逻辑变量，若结点 i 已向结点j 发送了一个标号信息但尚未从j 收到应答消息时值为"假"。

begin

根结点 s 的初始化算法：

Labeled(s) ← false；

send INIT to itself；

一般结点 i 的初始化算法：

Labeled(i) ← false；

在结点 i 处的算法：

upon receiving INIT message **do**：

  Labeled(i) ← true； Father(i) ← i； Level(i) ← 0；

  Unvisited(i) ← Neighbour(i)； Sons(i) ← φ；

```
    if Unvisited(i) ≠ Φ
      then send LABEL(i, Level(i)) message to j, for all j∈Unvisited(i);
           Echoed(j) ← false
      else the algorithm terminated
    endif;
  upon receiving LABEL(k, lev) from sender k do:
    if Labeled(i)
       then if Father(i) = k
                then send LABEL(i, Level(i)) to j, for all j∈Unvisited(i);
                     Echoed(j) ← false
                else  send ECHO(i, 'stop' ) to node k
             endif
       else  Labeled(i) ← true; Unvisited(i) ← Neighbour(i) − {k}; Sons(i) ← Φ;
             If Unvisited(i) = Φ
                then send ECHO(i, 'end' ) to Father(i)
                else send ECHO(i,  'keepon' )  to Father(i)
             endif
    endif;
  upon receiving ECHO(k, Status) message from sender k do:
    Echoed(k) ← true;
    if Status =  'keepon'   then Sons(i) ← Sons(i) ∪{k}
else if status =  'stop'
                                   then Unvisited(i) ← Unvisited(i) − {k}
                                endif
    endif;
    if Status =  'end'
        then Sons(i) ← Sons(i)  ∪{k};
             Unvisited(i) ← Unvisited(i) − {k}
    endif;
    if Unvisited(i) ≠ Φ
       then if Echoed(j): for all j∈ Unvisited(i)
                then  if Father(i) = i
                         then send LABEL(i, Level(i)) to j, for all j;
                              Echoed(j) ← false
                         else send ECHO(i, 'keepon' ) to Father(i)
                      endif
                else skip
             endif
       else  if Father(i) ≠ i
```

then send ECHO(i, 'end') to Father(i)

else The algorithm terminated

endif

endif

end .

3\. **算法的正确性证明**

引理 3.3 在算法 3.5 终止时，每个结点都给予了标号，且每个结点有唯一的父结点，根是自己的父结点，这意味着算法的输出是一棵 BFS 树.

证明 对根结点 s 来讲，它标号自己且置其父结点就是自己。对任何一个结点 $i(\neq s)$，若它没有标号，则它总是其邻接结点中一个未标号的邻接结点．由于图 $G(V,E)$ 是连通的，且每一个标号结点都试图给未标号的邻接结点标号，所以结点 i 最终至少收到一个 LABEL 消息并获得一个父结点．此外，只有接收到 LABEL 消息才赋予其父结点，在整个算法执行过程中，每个结点也仅执行一次赋予自己父结点的操作，故每个结点至多只有一个父结点。

引理3.4 算法3.5最终将终止。

证明 根据观察：每个结点将被标号．且每个标号结点 v 对 LABEL 消息的响应是：若消息的发送者是非父的邻接结点，则 v 用 ECHO(v, 'stop') 响应；若发送者是父结点，v 最终用 ECHO(i, 'end') 消息响应．这样，在一个结点内，每个 LABEL 消息都由状态为 stop 或 end 的应答消息响应，这使得 Unvisited(v) 集合最终成为空集合．特别是在根结点 s 处，这导致整个算法的完成．

4\. **算法的复杂性分析**

定理 3.9 在异步通信的分布式计算模型上，对无向连通图 $G(V,E)$ 执行宽度优先搜索的算法 3.5，所需的通信复杂性和时间复杂性均为 $O(n^2)$.

证明 因为算法 3.5 是由许多次循环组成的．设第 i 次循环时，已构成的 BFS 树有 n_i 个结点 $(i \leqslant n_i \leqslant n)$，其中叶结点数为 $r_i(1 \leqslant r_i < n_i)$，叶结点度数之和为 $m_i(1 \leqslant m_i < m)$，树高为 $d_i(1 \leqslant d_i \leqslant n)$．则在第一阶段，树根发送 LABEL 消息至树叶的消息数为 $O(n_i)$，所需的时间为 $O(d_i)$。由叶发 LABEL 消息到根收到应答的消息数为 $(m_i - r_i)$，时间为 $O(1)$．在第二阶段，消息返回根的通信复杂性为 $O(m_i + n_i)$，时间复杂性为 $O(d_i + 1)$。故整个算法的通信复杂性为

$$\sum_{i=1}^{d} c(m_i + n_i - r_i) \leqslant \sum_{i=1}^{d} c(m_i + n) = O(dn) + c\sum_{i=1}^{d} m_i = O(n^2 + m) = O(n^2)$$

这里 $c>0$ 常数，d 为从 s 出发到达最远结点的最短路径 $(d<n)$．算法的时间复杂性为

$$c'\sum_{i=1}^{d}(2d_i + 1) = c'\left(2\sum_{i=1}^{d} d_i + d\right) \leqslant 2c'(d^2 + d) = O(d^2) = O(n^2)$$

c' 为正常数.

3.3 小　　结

本章重点讨论了在并行及分布式环境中对图进行遍历搜索的各种搜索算法．基于 SIMD－CREW PRAM，给出了 p－深度优先、p－宽度优先以及 p－宽深优先的搜索算法并分析了它们的计算复杂性，其中 p 是处理器数目 ($p \leqslant n$)。基于异步通信的分布式计算模型，本章给出了纯遍历搜索算法、DFS 搜索算法以及改进的 DFS 搜索算法．最后还介绍了两个 BFS 搜索算法，这两个算法各有特色，前者虽然具有最优的时间复杂性，但通信复杂性差；后者使通信复杂性有所改善，但以花费额外时间为代价．本章介绍的几种搜索技术是后面许多图论算法要用到的基本技术．

由于深度优先搜索本身固有的顺序性[2]，到目前为止，对一般图 $G(V,E)$ 进行深度优先搜索，尚未发现一个时间为 $O(\log^c n)$，处理器数为 $O(f(n))$ 的并行算法，其中 c 为大于零的常量，$f(n)$ 为 n 的一个非零多项式，因此这是一个仍未解决的问题．但 Smith 基于 SIMD－CREW PRAM，对平面图给出了第一个深度优先搜索的并行算法。他的算法需 $O(\log^3 n)$ 时间和 $O(n^4)$ 处理器[3]。最近 He 等人对 Smith 算法进行了改进，基于 SIMD－CREW PRAM，建议了一个简单的深度优先搜索并行算法，他们的算法执行时间为 $O(\log^2 n)$，仅需 $O(n)$ 处理器[4]．Kim 等人基于 SIMD－CREW PRAM，运用了求最短路径的并行算法，对无环有向图给出了一个 $O(\log^2 n)$ 时间和 $O(n^3/\log n)$ 个处理器的深度优先搜索的并行算法[5]．

在图的宽度优先搜索的并行算法中、大部分算法都是基于求图的最短路径。在 SIMD－CREW PRAM 上，已有的宽度优先搜索的并行算法的时间复杂性为 $O(\log^2 n)$、处理器数目复杂性为 $O(n^3/\log n)$[5,6]．因后面我们还要介绍求最短路径的并行算法，故宽度优先搜索的其它并行算法本章不再介绍．

值得一提的是：在过去的几年里，人们一直很重视分布式 BFS 算法的研究．1985 年 Awerbuch 使用了一种在异步环境中模拟同步环境算法的同步器 (Synchronizer) 技术，给出了一个通信复杂性为 $O(kn^2)$、时间复杂性为 $O(n\log_k n)$ 的分布式 BFS 算法 ($1 \leqslant k < n$)[13]．Frederickson 考虑了稀疏图这一特殊情况下的 BFS 算法，建议了一个通信和时间复杂性均为 $O(nm)$ 的算法[12]．尤其应该指出的，Awerbuch 等曾独立地建议了算法 3.5，利用分块 (Partitioning) 方法，在每个块之间执行算法 3.5，最后得到一个通信复杂性为 $O((m+n^{1.5})\log n)$，时间复杂性为 $O(n^{1.5}\log n)$ 的分布式算法．还进一步指出，若过程递归调用，则分布式 BFS 算法的期望通信复杂性为 $O(m2^{\sqrt{\log n \log\log n}})$，时间复杂性为 $O(n2^{\sqrt{\log n \log\log n}})$[14]．由于这些 BFS 算法较这里介绍的算法复杂得多，故未曾介绍，有兴趣的读者可去参阅有关文献．最近作者利用算法 3.5 的思想，在分布式计算模型上，对二分图最大基数匹配问题建议了一个通信复杂性为 $O(n^{1/2}(n^2+m))$，时间复杂性为 $O(n^{5/2})$ 的算法[15]。

参 考 文 献

[1] Reghbati E, Corneil D. Parallel Computations in Graph Theory, *SIAM J. Comput.*, 7(2), 1978, 230–237

[2] Reif J. Depth First Search is Inherently Sequential, Aiken Computation Lab Technical Report TR–27–38, Haravard University, Cambridge, 1983

[3] Smith J. Parallel Algorithms for Depth–First Searches I: Planar Graphs, *SIAM J. Comput.*, 15(3), 1986, 814–830

[4] He X, Yesha Y. A Nearly Optimal Parallel Algorithm for Constructing Depth First Spanning Trees in Planar Graphs, *SIAM J. Comput.*, 17(3), 1988, 486–491

[5] Kim T, Chwa K. Parallel Algorithms for a Depth First Search and a Breadrh First Search, *Intern. J. Comput., Math.*, 19, 1986, 39–54

[6] Ghosh R K, Bhattacharjee G P. Parallel breadth first search algorithms for trees and graphs, *Intern. J. Comput., Math.*, 15, 1984, 255–268

[7] Chenng T Y. Graph Traversal Techniques and the Maximum Flow Problem in Distributed Computation, *IEEE Trans. Soft. Eng.*, SE–9(4), 1983, 504–512

[8] Chang E J H. Echo Algorithms: Depth Parallel Operations on General Graphs, *IEEE Trans. Soft. Eng.*, SE–8, 1982, 391–401

[9] Awerbuch B. A New Distributed Depth–First–Search Algorithm, *Inform. Proc. Lett.*, 20, 1985, 147–150

[10] Zhu Y, Cheung T Y. A New Distributed Breadth–First–Search Algorithm, *Inform. Proc. Lett.*, 25, 1987, 329–333

[11] Lakshmanan K B, Meenakshi N, Thulasiraman K. A Time–Optimal Message–Efficient Distributed Algorithm for Depth–First–Search, *Inform. Proc. Lett.*, 25, 1987, 103–109

[12] Frederickson G. A Single Source Shortest Path Algorithm for a Planar Distributed Network, *Proc. 2nd Simp. on Theoretical Aspects of Computer Science*, 1985

[13] Awerbuch B. Complexity of Network Synchronization, *J. ACM*, 32(4), 1985, 804–823

[14] Awerbuch B, Gallager R G. Distributed BFS Algorithms, *in Proc. 26th IEEE FOCS*, 1985, 250–266

[15] 唐策善，梁维发．二分图最大匹配问题的分布式算法，计算机工程与应用，(10–11)，1990，48–53



第四章　求连通分支的并行算法

寻找无向图连通分支是近十几年来并行图论算法研究领域一个最活跃的分支。无向图 $G(V,E)$的一个连通分支是 G 的一个最大连通子图，在这个子图中任意两个顶点之间都有可达路径。找出 G 的所有连通分支称之为找连通分支算法。目前在并行环境中，求图的连通分支有三种方法。第一种是采用某种形式的搜索技术，如 BFS 等；第二种是通过图的邻接矩阵计算图的传递闭包；最后一种是 Hirschberg 提出的顶点倒塌（Vertex Collapse）法。本章仅介绍后两种方法，尤其是最后一种方法在各种并行计算模型上的实现。

4.1　传递闭包法

Reghbati 等人[5]曾建议了用图 G 的自反传递闭包求图的连通分支。他们的算法将描述在下面。约定 $G(V,E)$ 的邻接矩阵是 A，A 的自反传递闭包是 B，那么 B 的元素定义如下:

$$b_{ij}=\begin{cases}1, & \text{顶点 } i \text{ 与顶点 } j \text{ 之间有路径存在;}\\0, & \text{否则.}\end{cases}$$

第三个矩阵 C 是通过矩阵 B 构造出来的，C 的元素定义如下:

$$c_{ij}=\begin{cases}j, & \text{若} b_{ij}=1;\\\infty, & \text{否则.}\end{cases}$$

同时还约定每个连通分支有相同的标识，我们把每个连通分支中最小标号顶点的标号作为它所在连通分支的标识，设顶点 $i\in V$ 的连通分支标识为 $D(i)$。

根据自反传递闭包定义 $B=(A+I)^n$，其中 I 为单位矩阵，符号“+”定义为逻辑加，则求连通分支的详细算法形式化描述如下:

算法4.1　CONNECTED－COMPONENTS USING TRANSITIVE－CLOSURE

```
procedure Connected_Components (G,V,E);
begin
  (1) for each i, j : 1≦i, j ≦n pardo    / * 初始化 * /
          D(i)←i;
          b(i,j)←a(i,j)
      endfor;
    for l←1 to ⌈logn⌉ do    / * 步骤(2)重复 ⌈logn⌉次 * /
  (2) for each i, j, k : 1≤i, j, k≤n pardo    / * 计算B * /
```

```
            c'(i, j, k) = b(i, k)∧b(k, j);
            b(i, j) = ∨_{k=1}^{n} c'(i, j, k)
      endfor
    endfor;
  (3) for each i, j: 1 ≤ i, j ≤ n pardo    /* 计算C */
        if b(i,j) = 1 then c(i,j) ← j
                      else c(i,j) ← ∞
        endif
      endfor;
  (4) for each i : 1 ≤ i ≤ n pardo    /* 求连通分支标识 */
        for each j : 1 ≤ j ≤ n pardo
          D(i) ← min {c(i,j) | c(i,j) ≠ ∞}
        endfor
      endfor
end.
```

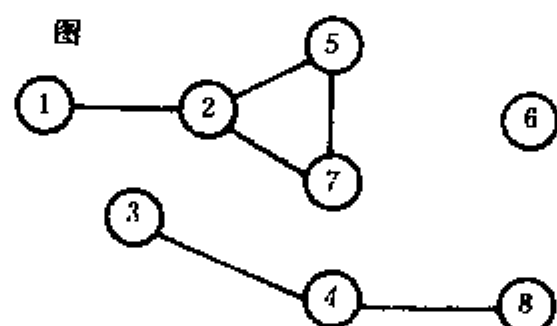
引理 4.1　在 SIMD－CREW PRAM 上，计算一个 n 阶矩阵的自反传递闭包的算法 4.1 需 $O(\log^2 n)$ 时间和 $O(n^3)$ 处理器。

证明　矩阵 A 的自反传递闭包 $B=(A+I)^n$，故 B 的计算可变成计算 $(A+I)$，$(A+I)^2$，$(A+I)^4$，…，$(A+I)^{2^{\lceil \log n \rceil}}$ 这一序列求得。也即对 $(A+I)$ 进行 $\lceil \log n \rceil$ 次矩阵自乘求得。由第 (2) 步可知：两个矩阵相乘，若用 n^3 个处理器，则可在 $O(\log n)$ 时间完成，所以求一个矩阵的自反传递闭包需 $O(\log^2 n)$ 时间和 $O(n^3)$ 处理器。

定理 4.1　在 SIMD－CREW PRAM 上，计算无向图 $G(V, E)$，$|V|=n$ 的连通分支的算法 4.1 需 $O(\log^2 n)$ 时间和 $O(n^3)$ 处理器。

证明　从算法 4.1 可知：算法的第 (1)，(3) 步均需 $O(1)$ 时间和 $O(n^2)$ 处理器；算法的第 (2) 步需 $O(\log^2 n)$ 时间和 $O(n^3)$ 处理器；算法的第 (4) 步需 $O(\log n)$ 时间和 $O(n^2)$ 处理器。故整个算法需 $O(\log^2 n)$ 时间和 $O(n^3)$ 处理器。

图4.1例示了此算法大概的执行过程。



A	1	2	3	4	5	6	7	8
1	1	1	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0
3	0	0	1	1	0	0	0	0
4	0	0	1	1	0	0	0	0
5	0	1	0	0	1	0	1	0
6	0	0	0	0	0	1	0	0
7	0	1	0	0	1	0	1	0
8	0	0	0	1	0	0	0	1

B	1	2	3	4	5	6	7	8
1	1	1	0	0	1	0	1	0
2	1	1	0	0	1	0	1	0
3	0	0	1	1	0	0	0	1
4	0	0	1	1	0	0	0	1
5	1	1	0	0	1	0	1	0
6	0	0	0	0	0	1	0	0
7	1	1	0	0	1	0	1	0
8	0	0	1	1	0	0	0	1

C	1	2	3	4	5	6	7	8
1	1	2	∞	∞	5	∞	7	∞
2	1	2	∞	∞	5	∞	7	∞
3	∞	∞	3	4	∞	∞	∞	8
4	∞	∞	3	4	∞	∞	∞	8
5	1	2	∞	∞	5	∞	7	∞
6	∞	∞	∞	∞	∞	6	∞	∞
7	1	2	∞	∞	5	∞	7	∞
8	∞	∞	3	4	∞	∞	∞	8

顶点	连通分支标识
1	1
2	1
3	3
4	3
5	1
6	6
7	1
8	3

图 4.1　利用传递闭包寻找连通分支

值得一提的是：Kucera 基于 SIMD－CRCW PRAM，建议了一个用自反传递闭包求连通分支算法[7]。他的算法的关键点是：使用 $O(n^2)$ 处理器在常量时间内求出 n 个元素的最小值，具体做法如下：

假定我们要找 c_1，c_2，…，c_n 这 n 个元素的最小值，使用 n^2 个处理器，处理器编号采用阵列形式，约定 PE_{ij} 表示编号为 (i,j) 处的处理器。n 个临时变量为 t_1，t_2，…，t_n。

算法 4.2　MINIMUM OF n ELEMENTS

begin

(1) 对所有 i，PE_{i1} 置 t_i 的值为0；

(2) 若 $c_i < c_j$，PE_{ij} 置 $t_j = 1$，在这一步结束时，$t_i = 0$ 当且仅当

$c_i = \min\{c_1, c_2, \cdots, c_n\}$；

/ * 这时可能存在多个最小值，第3步仅一个处理器返回最小值 * /

(3) 若 $t_i = 0$ 且 $i < j$，则 PE_{ij} 置 $t_j = 1$，此刻仅一个临时变量的值为0；

(4) 若 $t_i = 0$，PE_{i1} 返回 c_i 值作为最小值

end.

求最小值算法 4.2 可用来求最大值和布尔矩阵的乘法运算。例如，设布尔矩阵 A，B 的逻辑乘积为 C，则 $c_{ij} = \max\{a_{i1} \wedge b_{1j}, a_{i2} \wedge b_{2j}, \cdots, a_{in} \wedge b_{nj}\}$，它可用 n^2 个处理器在 $O(1)$ 时间内完成 $(1 \leqslant i, j \leqslant n)$。故在使用 n^4 处理器时计算矩阵 C 可在 $O(1)$ 时间内完成。

定理 4.2　在 SIMD－CRCW PRAM 上，求无向图 $G(V,E)$，$|V| = n$ 的所有连通分支的

算法 4.1 需 $O(\log n)$ 时间和 $O(n^4)$ 处理器.

证明 执行一次矩阵乘法运算需 $O(1)$ 时间和 $O(n^4)$ 处理器，而由图的邻接矩阵计算自反传递闭包需执行 $\lceil \log n \rceil$ 次矩阵乘法运算，由算法 4.1 可知在 SIMD－CRCW PRAM 上计算无向图连通分支需 $O(\log n)$ 时间和 $O(n^4)$ 处理器.

4.2 顶点倒塌法

4.2.1 算法的基本原理

Hirschberg 首先提出用顶点倒塌法求图的连通分支[1,2]. 算法的输入是邻接矩阵 A，为标记每一个连通分支，将每个连通分支内最小标号顶点作为该连通分支标识. 约定 $i \in V$，用 $D(i)$ 表示 i 所在的连通分支标识. 算法结束时，具有相同连通分支标识的顶点位于同一连通分支内. 算法的基本思想是：每个顶点仅是一个超顶点 (Supervertex) 的成员，而每个超顶点都是由最小标号顶点标识，这个顶点叫做“根”. 算法开始时，每个顶点都是它自己这个超顶点的根. 并行算法是由一系列循环完成的. 而每次循环又分为三步：第一步，发现每个顶点的最小标号邻接超顶点；第二步，把每个超顶点的根连到最小标号邻接超顶点的根上；第三步，所有在第二步连接在一起的超顶点倒塌成一个较大的超顶点. 因为超顶点的个数每次循环后至少减少一半，所以把每个连通分支倒塌成为单个超顶点至多 $\lceil \log n \rceil$ 次循环即可.

4.2.2 算法的形式化描述

算法 4.3 CONNECTED－COMPONENTS USING COLLAPSE VERTICES

输入：邻接矩阵A；

输出：向量 $D(1:n)$，其中 $D(i)$ 表示顶点 i 的标识.

```
begin
  (1) for each i : 1 ≤ i ≤ n pardo    /* 初始化 */
          D(i) ← i
      endfor;
      do step (2) through step (6)  ⌈logn⌉ iterations
  (2) for each i, j : 1 ≤ i, j ≤ n pardo   /* 找邻接顶点中的最小超顶点 */
          C(i) ← min_j {D(j) | A(i, j) = 1 and D(i) ≠ D(j)};
          if none then C(i) ← D(i) endif
```

```
        endfor ;
    (3) for each i, j : 1≤ i, j ≤ n pardo    / * 求每个超顶点的最小邻接超顶点 * /
                C(i) ← min {C(j) | D(j) = i and C(j) ≠ i} ;
                if none then C(i) ← D(i) endif
        endfor ;
    (4) for each i : 1 ≤ i ≤ n pardo
                D(i) ← C(i)
        endfor ;
    (5) for ⌈logn⌉ iterations do    / * 找每个顶点新的超顶点 *
            for each i : 1 ≤ i ≤ n pardo
                C(i) ← C(C(i))
            endfor
        endfor ;
    (6) for each i : 1 ≤ i ≤ n pardo
                D(i) ← min {C(i), D(C(i))}
        endfor
end .
```

4.2.3 算法的正确性证明

在给出算法的正确性证明之前，首先我们引入一个称为 k— 树环(k—tree—loop)的概念。一个有向图是一个 k—树环($k \geqslant 0$) 当且仅当它的每个顶点的出度都为 1，且在此图中存在唯一的一个长度为 k+1 的有向回路。注意一个 k—树环至少含有 k+1 个顶点。

树环是指对某个 $k(k \geqslant 0)$ 的 k— 树环。在 $k = 0$ 这一特殊情况下，我们定义 0－ 树环的根为一个顶点 v，它是由一条边 (v,v) 组成，长度为 1 的有向回路。一个俱乐部 (Club) 是一个 0－ 树环，在此树环中所有边都指向根顶点，图 4.2 说明了这几个定义。

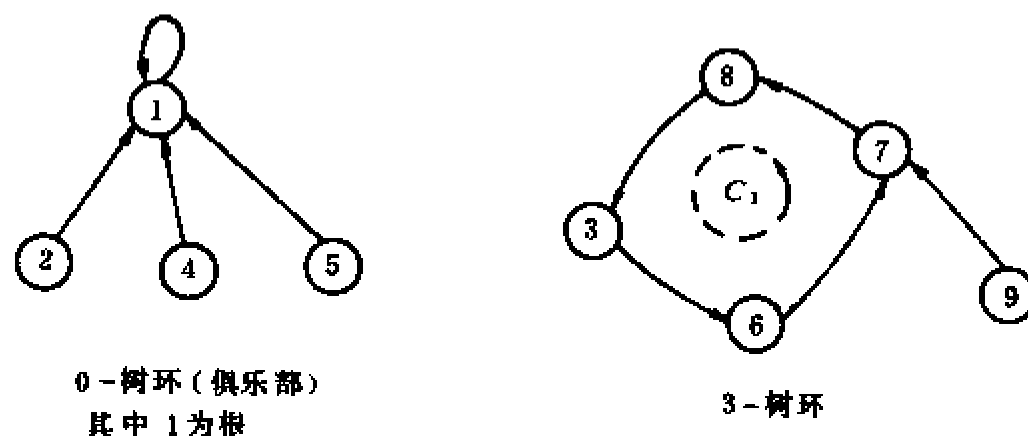


图 4.2 k—树环

引理 4.2 令 $G_c = (V_c, E_c)$ 是 $G(V, E)$ 的一个连通分支且 $|V_c| \geqslant 2$，定义一个函数 c: $V_c \to V_c$，其中 $c(i) = \min\{j \mid A(i,j) = 1 \text{ and } j \neq i\}$，函数 c 定义了一个有向图 $G_c(c) = (V_c, E'')$，这里 $E'' = \{ <i, c(i)> \mid i \in V_c\}$，那么 $G_c(c)$ 是由许多 1— 树环组成

的，而且在每个树环中最小标号顶点位于树环的有向回路上。

证明 根据 c 函数的定义，$G_c(c)$ 是由树环组成的。由于 $c(i)\neq i$, 故 $G_c(c)$ 中不存在 0—树环。若 $G_c(c)$ 存在一个 k— 树环 $(k>0)$。令 $v_0,v_1,\cdots,v_k,v_0$ 表示 k— 树环中的有向回路（即 $c(v_i)=v_{i+1}$, $i=0, 1, \cdots, k-1$，且 $c(v_k)=v_0$）。不失一般性，令 $v_0=\min\{v_0,v_1,\cdots,v_k\}$。而在 $G_c(c)$ 中，v_1、v_2 都同 v_0 是邻接顶点且 $c(v_1)=v_2>v_0$。当 $k>1$ 时，这与前提矛盾，故 $G_c(c)$ 中仅存在 1— 树环。也就是说，v_0 与 v_1 组成有向回路 $c(v_0)=v_1$，$c(v_1)=v_0$。类似地我们可以证明树环上最小标号顶点也必须在树环的有向回路上。

根据 c 函数定义，下面我们给出函数 c^k 的定义。$c^k: V_c\rightarrow V_c$

$$c^k(v)=\begin{cases} c(v), & k=1 \\ c(c^{k-1}(v)), & k>1 \end{cases} \qquad \forall \quad v\in V_c$$

引理 4.3 令 v 是 $G_c(c)$的任意一个树环顶点，且令 v_0、v_1 是 u 所在树环的有向回路上的两个顶点，那么，对 $N\geqslant n-2$，$c^N(v)$及 $c^{N+1}(v)$这两个数中的一个等于 v_0，而另一个等于 v_1。

证明 因为从 v 到达 v_0 或 v_1 顶点的较近一个的距离至多为 $n-2$，故由这一事实，上述命题不难得证。

有了上述的诸定义及引理，现给出算法的正确性证明。

定理 4.3 算法 4.3 根据输入的对称邻接矩阵 A 可计算出无向图 $G(V,E)$的所有连通分支。

证明 对一个孤立顶点的平凡图，算法显然能计算出它的连通分支。下面我们证明 $|V|\geqslant 2$ 时，算法也能给出所有的连通分支。令 $G(D)=(V, E_D)$ 是由函数 D 定义的一个有向图，其中 $E_D=\{<i, D(i)>\mid i\in V\}$。在执行算法的第 (1) 步之后、第 (2) 步之前，$G(D)$ 满足下列一些性质:

(Ⅰ) $G(D)$是俱乐部集合;

(Ⅱ) 每个俱乐部的根是该俱乐部的最小标号顶点;

(Ⅲ) 任何一个俱乐部的顶点集都是某个连通分支顶点集的一个子集.

我们要证明: 若在执行第(2)步之前$G(D)$满足性质(Ⅰ)-(Ⅲ)，那么在执行第(2)-(6)步之后，新的函数 D(第(6)步计算的)导出的图 $G(D)$也有性质(Ⅰ)-(Ⅲ)。此外，若在执行第(2)步之前，每个连通分支内至少有两个俱乐部，则循环后俱乐部的个数至少减少一半。

首先我们注意第 (1) 次执行第 (2) – (6) 步的情形。开始时 D 是一个单位函数，第 (2) 步函数 c 的定义同引理 4.2 的定义一致，故建立 G 的每个连通分支的 1— 树环；第 (3) 步 c 没有改变，因为 c 改变仅当 $D(j)=i$ 且 $c(i)\neq i$; 在第 (4) 步，函数 c 拷贝到 D；而在第 (5) 步，c 置换为 c^N，这里 $N=2^{\lceil\log n\rceil}$; 第 (6) 步置 $D(i)$ 为 $\min\{c^N(i), D(c^N(i))\}$ 等同于置 $D(i)$ 为 $\min\{c^N(i), c^{N+1}(i)\}$。由引理 4.3 可知，对所有 $i\in V$，$D(i)$ 等于包含 i 的树环的最小标号顶点，这样每个树环的顶点集已归并到一个俱乐部中，故执行第 1 次循环后，$G(D)$ 仍满足性质 (Ⅰ) – (Ⅲ)。因为每个 1— 树环至少含有两个顶点，在每个非平凡的连通分支中，它的俱乐部数目不超过它原来数目的一半。

正如前面已提到的，进一步的循环将是归并超顶点（即俱乐部）．超顶点间连接定义如下：令 V_r 指定为 $G(D)$ 中所有俱乐部的根的集合。令 $G_r=(V_r,E_r)$ 是一个无向图，这里 $i\neq j$，$(v_i,v_j)\in E_r$ 当且仅当在 v_i 及 v_j 所在的俱乐部分别存在两个成员顶点 v_i'、v_j'，且 $(v_i', v_j')\in E$．换句话说，两个超顶点互为邻接超顶点当且仅当存在一条连接它们成员顶点的边．函数 c 在第 (2) 步和第 (3) 步建立．在第 (2) 步，每个顶点 i 审查它的邻接顶点的俱乐部成员资格，且置 $c(i)$ 为最小标号的邻接俱乐部标识。在第 (3) 步，每个 $i\in V_r$ 审查自己的俱乐部成员（由 $D(j)=i$ 指定），并从它所在俱乐部成员发现的最小标号俱乐部中挑选出一个最小的标号俱乐部标识自已。简单地讲，函数 $c: V_r\to V_r$ 是这样的一个函数，对所有 $i\in V_r$，$c(i)$ 等于在 G_r 中与 i 关联的最小标号的邻接顶点。正如引理 4.2 所说，c 在 G 上定义了许多 1— 树环．接下来我们将考虑 $i\notin V_r$ 的顶点，象这样的顶点，不存在满足 $D(j)=i$ 的 j，因而在第 (3) 步 $c(i)$ 重置为 $D(i)$，因此 $c: V\to V$ 在 G 上定义了许多 1— 树环。因为每个非根顶点都指向根，而根是 1— 树环。因而在执行第 (6) 步之后，新的函数 D 导出的 $G(D)$ 仍满足性质 (Ⅰ)—(Ⅲ)，此外，每个 1— 树环包含 G_r 中两个或多个顶点，即两个或多个俱乐部．因而在至少含有两个俱乐部的连通分支内，俱乐部数目至少减少一半。

由上述讨论可知，在每个连通分支仅由一个俱乐部组成之前，每次循环都使得俱乐部数目至少减半．不难证明，随后的循环都不影响单个俱乐部。又由于算法开始时，每个连通分支至多有 n 个顶点（俱乐部），故经过 $\lceil\log n\rceil$ 次循环足以把每个连通分支都归并到单个俱乐部内。D 定义了俱乐部的成员．

4.2.4 算法的复杂性分析

在分析算法的复杂性之前，先将算法 4.3 的第 (2) 步进一步细化。设 Temp(1:n; 1:n) 为一个临时数组，Index(i) 为一个下标变量。第 (2) 步细化如下：

```
(2a) for each i,j : 1 ≤ i,j ≤ n pardo
         if (A(i,j) = 1)∧(D(j)) ≠ D(i)
             then Temp (i,j) ← D(j)
             else Temp(i,j) ← ∞
         endif
     endfor ;
(2b) for k ← 0 to ⌈logn⌉ − 1 do
         for each i, j: 1 ≤ i, j ≤ n pardo
             if (j + 2^k) mod (n + 1) ≠ 0
                 then Index(i) ← (j + 2^k) mod (n + 1)
                 else Index(i) ← 1
             endif ;
             Temp(i,j) ← min{Temp(i, j), Temp (i, Index(i))}
```

```
            endfor
        endfor ;
   (2c) for each i : 1 ≤ i ≤ n pardo
            if Temp(i, 1) = ∞ then C(i) ← D(i) else C(i) ← Temp(i, 1) endif
        endfor ;
```

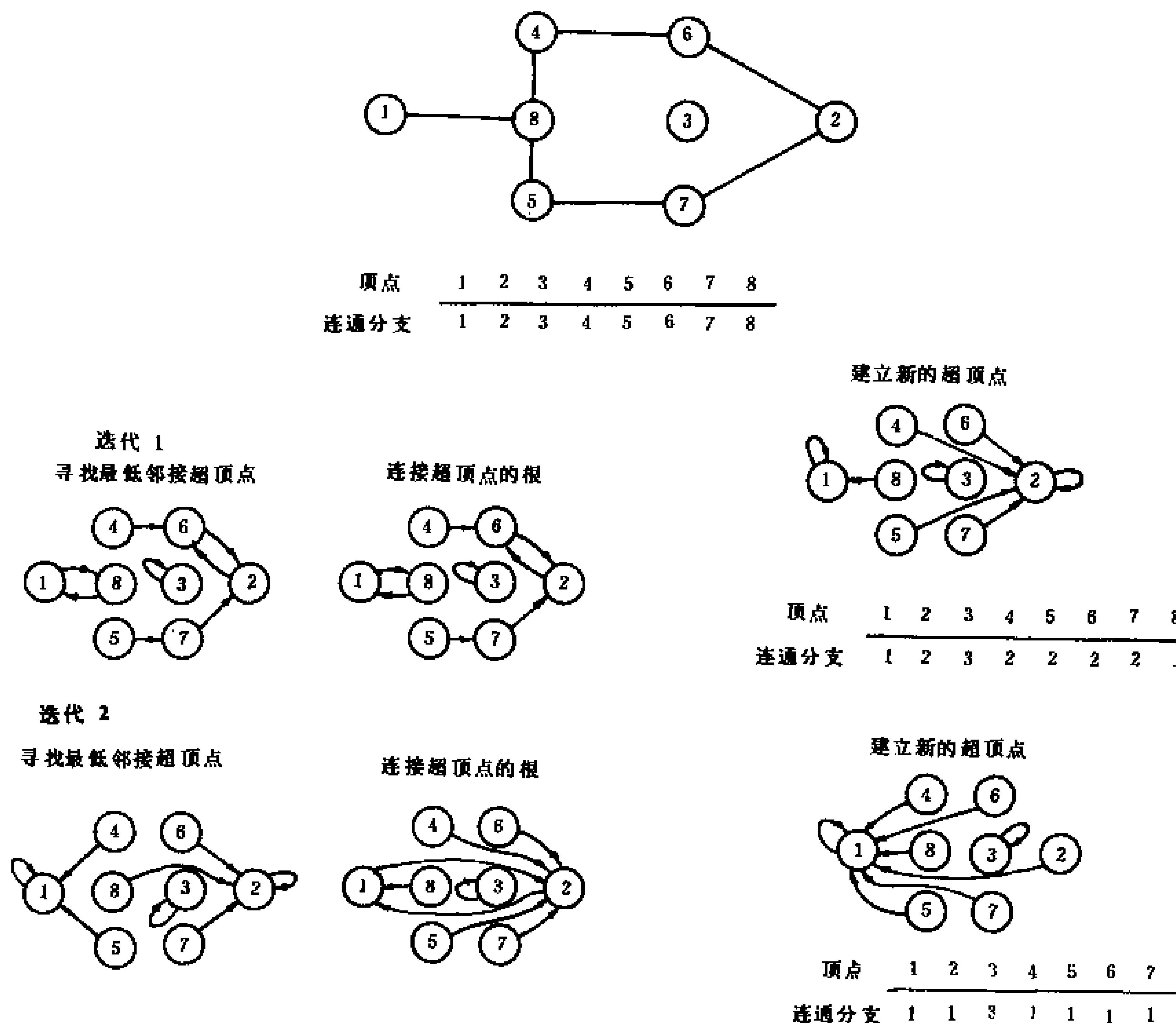


图 4.3 顶点倒塌法求连通分支

从第 (2) 步细化可以看出，实现此步需 $O(n^2)$ 处理器和 $O(\log^2 n)$ 时间。

定理 4.4 在 SIMD－CREW PRAM 上，求无向图 $G(V,E)$, $|V| = n$ 的所有连通分支的算法 4.3 需 $O(\log^2 n)$ 时间和 $O(n^2)$ 处理器。

证明 从算法 4.3 我们可以看到：算法的第 (1) 步、第 (4) 步以及第 (6) 步均可在 $O(1)$ 时间内完成，若使用 n 个处理器；算法的第 (3) 步同第 (2) 步一样地实现，这两步均需 $O(\log n)$ 时间和 $O(n^2)$ 处理器；又第 (2)、(6) 步需 $\lceil \log n \rceil$ 次循环，故整个算法需 $O(\log^2 n)$ 时间和 $O(n^2)$ 处理器。

推论 4.1 在 SIMD－CREW PRAM 上，求无向图 $G(V,E)$ 的所有连通分支需 $O(\log^2 n)$

时间和 $O(n^2/\log n)$ 处理器。

证明 因为对 n 个元素求最小值可采用分组技术。使用 $O(n/\log n)$ 处理器时，每个处理器最多分到 $\lceil\log n\rceil$ 个元素，每个处理器首先计算自己的最小值，这需要 $O(\log n)$ 时间。然后这 $O(n/\log n)$ 个处理器并行的从各自的最小值中选出全体元素的最小值，这一步需要 $\log(\lceil n/\log n\rceil) \leqslant \lceil\log n - \log\log n\rceil$ 时间，所以从 n 个元素中选取最小值只需要 $O(\log n)$ 时间和 $O(n/\log n)$ 处理器；把这种技术应用到算法的各步，不难推得算法需 $O(\log^2 n)$ 时间和 $O(n^2/\log n)$ 处理器。

图 4.3 表示顶点倒塌算法的执行过程。

4.3 最优的连通分支算法

Chin 等人在研究算法 4.3 的基础上，基于同一计算模型给出了一个改进算法[3,4]。改进后的算法是一个最优算法。他们的算法同顶点倒塌算法 4.3 的不同之处在于：在每次循环的归并过程中，仅限于对超顶点进行操作，而不再对已被归并的超顶点以及孤立的超顶点进行操作，这也是减少使用处理器的原因。为此他们在算法中引入了一个长度为 n 的 flag 向量以及清扫步（算法的第(7)，(8)步）来标识超顶点以便及时地清扫那些已被归并的超顶点和那些已成为孤立的超顶点。其中：flag(i)＝1 表示顶点 i 是超顶点；flag(i)＝0 表示顶点 i 已归并到某个超顶点或 i 是一个孤立超顶点。在算法的其后循环过程中，flag(i)＝0 的顶点 i 将不再考虑。他们算法的形式化描述如下：

算法 4.4 OPTIMAL CONNECTED—COMPONENTS ALGORITHM

输入：n 阶邻接矩阵 A；

输出：向量 $D(1:n)$，其中 $D(i)$ 表示顶点 i 的连通分支标识，它等于 i 所在连通分支的最小标号顶点标识。

```
begin
  (1) for each i : 1 ≤ i ≤ n pardo    /* 初始化 */
          D(i) ← i;     flag(i) ← 1
      endfor;
   do step (2) throrgh step (8)  ⌈log n⌉ iterations
  (2) (2a) for each  i : 1 ≤ i ≤ n  pardo    /* 找出非孤立顶点集 S */
                S ← { i | flag(i) = 1}
           endfor ;
      (2b) for each i, j : i , j∈S pardo    /* 找每个超顶点的邻接顶点 */
               c(i) ← min{j |A(i,j ) = 1} ;
                      j∈S
               if none then c(i) ← i endif
           endfor ;
  (3) for each i : i∈S pardo
```

```
        if c(i) = i then flag(i) ← 0  endif   /* 删除孤立超顶点 */
    endfor;
(4) for each i: i∈S pardo
        D(i) ← c(i)
    endfor;
(5) for ⌈logn⌉ iterations do   /* 求每个超顶点的顶点标识 */
        for each i: i∈S pardo
            c(i) ← c(c(i))
        endfor
    endfor;
(6) (6a) for each i: i∈S pardo
             D(i) ← min{c(i), D(c(i))}
         endfor;
    (6b) for each i: 1 ≤ i ≤ n pardo
             D(i) ← D(D(i))
         endfor;
    /* 清扫步 */
(7) (7a) for each i, j: (i∈S) ∧ (j∈S) (j = D(j)) pardo
             A(i,j) ← OR_{k∈S} {A(i,k) | D(k) = j}
         endfor;
    (7b) for each i, j: (i∈S) ∧ (j∈S) ∧ (j = D(j)) ∧ (i = D(i))  pardo
             A(i,j) ← OR_{k∈S} {A(k,j) | D(k) = i}
         endfor;
    (7c) for each i: i∈S pardo
             A(i,i) ← 0
         endfor;
(8) for each i: i∈S pardo   /* 形成新的超顶点集 */
            if D(i) ≠ i then flag(i) ← 0 endif
    endfor
end.
```

算法的正确性在上面已经给予了证明，下面我们只对它的复杂性进行分析。首先我们引入两个引理。

引理 4.4 已知 n 个元素的集合 $\{a_0, a_1, \cdots, a_{n-1}\}$ 以及 K 个处理器，那么表达式 $A(n) = a_0 * a_1 * a_2 * \cdots * a_{n-1}$ 可在 $T(n,K)$ 个单位时间内计算出来，这里“$*$”是任意一个二元结合运算符，且

$$T(n,K)=\begin{cases}\lceil n/K\rceil-1+\log K, & 若\lfloor n/2\rfloor>K,\\ \log n, & 否则。\end{cases}$$

证明 若 $K\geqslant\lfloor n/2\rfloor$，则利用递归折叠(Recursive Doubling)技术在 $\log n$ 时间内可将 $A(n)$ 计算出来。现考虑 $K<\lfloor n/2\rfloor$ 情形。我们将 $\{a_0,\cdots,a_{n-1}\}$ 划分为 K 个组，除最后一组外，每组有 $\lceil n/K\rceil$ 个元素，最后一组有 $r=n-(K-1)\lceil n/K\rceil$ 个元素。每组分配一个处理器并计算各组的部分子积，完成这一任务需 $\lceil n/K\rceil-1$ 个单位时间，结果得到 K 个部分子积。然后 K 个处理器合作，并行计算 K 个部分子积的乘积，最后算出 $A(n)$，完成这后面的任务需 $\log K$ 时间。故一共需时间为 $\lceil n/K\rceil-1+\log K$。

引理 4.5 在 SIMD－CREW PRAM 上，已知有 nK 个处理器，若 $1\leqslant K\leqslant\lfloor n/2\rfloor$，则算法 4.4 的 (2$b$) 步至多需要 $O(n/K+\log n\log K)$ 时间完成，否则只需要 $O(\log^2 n)$ 时间就可完成。

证明 由算法 4.4 可知，若 $1\leqslant K<\lfloor n/2\rfloor$，则每次循环后超顶点数至少减少一半，即每次循环后 $|S|$ 减少一半。这样在执行了 $t=\lceil\log n\rceil-\log K$ 次循环之后，$|S|$ 至多为 $2K$，因此得出时间上界 $T_1(n,K)$ 为：

$$T_1(n,K)=\sum_{k=0}^{t-1}\left(\lceil\lfloor n/2^k\rfloor/K\rceil-1+\log K\right)+\sum_{k=t}^{\lceil\log n\rceil-1}\log(n/2^k)$$

$$\leqslant\lceil 2n/K\rceil+t\log K+(\log K)^2$$

$$\leqslant O(n/K+\log n\log K)$$

若$K\geqslant\lfloor n/2\rfloor$，则时间上界为：

$$T_1(n,K)=\sum_{k=0}^{\lceil\log n\rceil-1}\log(n/2^k)=O(\log^2 n)$$

类似地可以证明算法4.4的第(7)步，若使用nK个处理器，也可在$T_1(n,K)$时间内完成。

定理 4.5 在 SIMD－CREW PRAM 上，求一无向图 $G(V,E)$ 的连通分支算法 4.4 需 $O(n/k+\log^2 n)$ 时间和 $O(nK)$ 处理器 $(K\geqslant 1)$。尤其当 $K=n/\log^2 n$ 时，这时算法需 $O(\log^2 n)$ 时间和 $O(n^2/\log^2 n)$ 处理器。

证明 首先我们计算一下每步的执行时间以及需要的处理器数目。

步	总的计算时间		处理器数
	$1\leqslant K<\lfloor n/2\rfloor$	$K\geqslant\lfloor n/2\rfloor$	
(1)	$O(1)$	$O(1)$	n
(2a)	$O(\log^2 n)$	$O(\log^2 n)$	n
(2b)	$O(n/K+\log n\log K)$	$O(\log^2 n)$	nK
(3)	$O(\log n)$	$O(\log n)$	n
(4)	$O(\log n)$	$O(\log n)$	n
(5)	$O(\log^2 n)$	$O(\log^2 n)$	n
(6)	$O(\log n)$	$O(\log n)$	n
(7)	$O(n/K+\log^2 n)$	$O(\log^2 n)$	nK
(8)	$O(\log n)$	$O(\log n)$	n

故整个算法需 $O(n/K+\log^2 n)$时间和 $O(nK)$处理器。

4.4 稀疏图的连通分支算法

在前几节介绍的算法对稠密图（边较多的图）来讲是有效的，但对稀疏图（边较少的图如 $E=O(n)$）未必是好的算法。为此 Savage 等人对算法 4.3 进行了一些修改，给出了稀疏图的连通分支算法[6]。

Savage 等人的算法同算法 4.3 的不同之处首先表现在算法的输入方面。她们算法的输入是由邻接表组成的矩阵 R，R 是一个 $n\times(n-1)$ 的矩阵，其中第 i 行是由顶点 i 的邻接顶点组成的 $(1\leqslant i\leqslant n)$。并且她们还用了一个辅助向量 $EM(1:n)$，这里 $EM(i)$ 是 R 的第 i 行最后一个元素的下标 $(1\leqslant i\leqslant n)$。对照算法 4.3，当输入为邻接表时，观察它是如何执行的？下面就讨论算法 4.3 在邻接表输入时的实现细节。

已知 i，我们可以用 $EM(i)$ 个处理器在 $O(1)$ 时间内找出集合 $\{j\mid A(i,j)=1\}=\{j\mid j$ 在 R 的第 i 行 $\}$ 的所有元素。因此算法 4.3 的第 (2) 步可使用 $O(\sum_{i\in V}EM(i))=O(m)$ 处理器在 $O(\log n)$ 时间内实现。

算法 4.3 的第 (3) 步实现起来似乎要难一点，因为 $\{j\mid D(j)=i\}$ 对所有 i 来讲似乎需要 $O(n^2)$ 处理器。注意，若 i 不是一个俱乐部的根，那么这一步 $D(i)$ 的值没有改变。若 i 是根，则集合 $\{j\mid D(j)=i\}$ 是根为 i 的俱乐部顶点的集合。为减少处理器数目，她们的算法是，令 B 是 D 的一个拷贝，令 I 是 $I(i)=i$ 的 n 元向量 $(1\leqslant i\leqslant n)$，用 Preparata 的排序算法[11] 对 B 进行排序。注意在排序过程中，若 $B(i)$ 与 $B(j)$ 交换了位置，则 $I(i)$ 也与 $I(j)$ 交换位置。排序需 $O(\log n)$ 时间和 $O(n\log n)$ 处理器，这时我们注意到，每个俱乐部的顶点在 I 中都聚在一起了。不难看出：在 $O(\log n)$ 时间内，使用 $O(n)$ 处理器同样可以修改 R 的一个拷贝 R'，使得 R' 的第 i 行是由根为 i 的俱乐部的顶点所组成。我们可用这个 R' 去实现算法 4.3 的第 (3) 步。基本方法同第 (2) 步一致。所以第 (3) 步需要 $O(\log n)$ 时间和 $O(m+n\log n)$ 处理器。

定理 4.6 在 SIMD－CREW PRAM 上，计算一个稀疏的无向图 $G(V,E)$，$|V|=n$，$|E|=m$ 的所有连通分支需要 $O(\log^2 n)$ 时间和 $O(m+n\log n)$ 处理器。

证明 根据上述的分析，在算法 4.3 的每一步实现过程中，最多需 $O(\log n)$ 时间和 $O(m+n\log n)$ 处理器，而算法第 (2)－(6) 步循环了 $\lceil\log n\rceil$ 次，故整个算法需 $O(\log^2 n)$ 时间和 $O(m+n\log n)$ 处理器。

4.5 一维阵列上的连通分支算法

到目前为止，我们介绍的连通分支算法都是基于理想的共享存贮模型，从本节开始，在后面的几节将介绍基于互连网络模型的连通分支算法。

计算无向图连通分支的一种简单方法是：首先每个顶点组成一个集合，其后，若存在

一条边把两个不同集合的元素联接起来，我们就把这两个集合归并为一个新的集合。当图的所有边都已检查完毕后，最后剩下的每个顶点集合对应图的一个连通分支。

Savage[10] 建议使用 $n+1$ 个处理器组成一维线性心动 (Systolic) 阵列。处理器 1，2，…，n，对应顶点 1，…，n。每个处理器 i 有两个域：第一个域包含顶点标号；第二个域 Comp(i)，表示顶点 i 的当前连通分支标号。初始状态时，这两个域的值都是 i。第 $n+1$ 处理器作为一个周转装置。一次只移入图的一条边到第一个处理器。奇（偶）数编号的处理器在奇（偶）数脉冲时激活，所以每隔一个脉冲就有一条边进入心动阵列，边记录从第一个处理器进入遍历直至第 $n+1$ 个处理器，然后再从第 $n+1$ 个处理器开始作反向遍历返回到第一个处理器，最后离开心动阵列，当处理完最后一条边记录时，对任何 $i \in V$，Comp(i) 就是 i 的连通分支标号了。

右移边记录 k 包括边 (u_k, v_k)，u_k 的当前连通分支标号 C_{u_k}，v_k 的当前连通分支标号 C_{v_k}；左移边记录 j 包括 (u_j, v_j)，$\min\{C_{u_j}, C_{v_j}\}$（用 $(C_{\min_j}$ 表示）及 $\max\{C_{u_j}, C_{v_j}\}$（用 $C_{\max_j}$ 表示）。

每个处理器 i 激活时执行的算法如下:

算法4.5 CONNECTED – COMPONENTS ON LINEAR ARRAY

procedure Computing_in_i_processor(i) ;

/ * $1 \leqslant i \leqslant n$，$i$为奇数时，仅在奇数脉冲时执行；反之 i 为偶数时，仅在偶数脉冲时执行。存于第$i-1$个处理器的(u_k, v_k)，C_{u_k}，C_{v_k} 进入第i个处理器以及存于第$i+1$个处理器的(u_j, v_j)，$C_{\min_j}$，$C_{\max_j}$ 也进入第i个处理器 * /

begin

  if Comp(i) $= C_{\max_j}$ **then** Comp(i) $\leftarrow C_{\min_j}$ **endif**;

  if $u_k = i$ **then** $C_{u_k} \leftarrow$ Comp(i)

    else if $v_k = j$ **then** $C_{v_k} \leftarrow$ Comp(j) **endif**

  endif;

  if $C_{u_k} = C_{\max_j}$ **then** $C_{v_k} \leftarrow C_{\min_j}$ **endif**;

  if $C_{vk} = C_{\max_j}$ **then** $C_{v_k} \leftarrow C_{\min_j}$ **endif**

end .

第 $n+1$ 个处理器具有特殊用途，它的作用是将边记录返回到心动阵列。其程序为

procedure Computing_last_processor ;

/ * 若$n+1$为奇数，仅在奇数脉冲时执行，反之在偶数脉冲时执行 * /

begin

  if $C_{u_k} < C_{v_k}$ **then** $C_{\min_k} \leftarrow C_{u_k}$； $C_{\max_k} \leftarrow C_{v_k}$

```
            else C_{min_k} ← C_{v_k};  C_{max_k} ← C_{u_k}
    endif
  end .
```

图4.4例示了此算法的执行过程。

定理 4.7 在一维线性心动阵列上，求一无向图 $G(V,E)$，$|V|=n$，$|E|=m$ 的所有连通分支需 $O(m+n)$ 时间和 $O(n)$ 处理器。

证明 因为图的每条边必须顺序地进入心动阵列，这需 $O(m)$ 时间，而每条边在心动阵列中驻留时间为 $O(n)$。故算法需 $O(m+n)$ 时间和 $O(n)$ 处理器。

4.6 二维网孔上的连通分支算法

Nassimi 等人[12] 已将算法 4.3 移植到网孔模型上。在他们的算法中，许多地方都要执行两个重要子过程，分别称之为 RAR(Random_Access_Read) 及 RAW(Random_Access_Write)[13]。过程 RAR 有两个参数，第一个参数是接收数据的地址，第二个参数是一个表达式，表示要读的值。例如在执行了 RAR$(a(i),b(c(i)))$ 之后，每个未屏蔽的处理器 i 中局部变量 $a(i)$ 的值等于处理器 $c(i)$ 中局部变量 $b(c(i))$ 的值。过程 RAW 也有两个参数，第一个参数表示要写的值，第二个参数是要写的地址。如在执行了 RAW$(a(i),b(c(i)))$ 之后，每个未屏蔽的处理器 i 中局部变量 $a(i)$ 的值将写入处理器 $c(i)$ 中局部变量 $b(c(i))$ 所在的单元中去。若有多个值往同一地址单元写，约定这些值中的最小值是最后的结果值。

为描述算法方便起见，引入函数 BITS，BITS(i,j,k) 返回整数 i $(i>0)$ 的二进制表示的第 j 位到 k 位的值 (0 位是最低位)。如

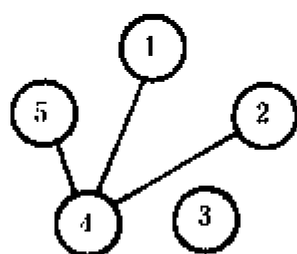
$$\text{BITS}(17,3,1)=0,\qquad \text{BITS}(10,3,2)=2,$$
$$\text{BITS}(16,4,4)=1,\qquad \text{BITS}(15,2,3)=0.$$

已知无向图 $G(V,E)$ 有 $n=2^k$ 个顶点，图 G 的度数为 d，令 adj(i,j) 是顶点 i 的邻接表 $(1\leqslant i\leqslant n,\ 1\leqslant j\leqslant d)$。如果顶点 i 有 $d_i(<d)$ 条关联边，那么对于 $d_i+1\leqslant j\leqslant d$ 有 adj$(i,j)\leftarrow\infty$。每个顶点的邻接表存放在对应编号处理器的局部存贮器中。假定 n 个处理器的二维网孔采用洗牌行主编号[14]。

在叙述详细算法之前，我们引入算法中要用到的一个过程 Reduce，此过程对应于算法 4.3 的第 (5) 步，将同一超顶点内的所有顶点都用根来标号。具体细节如下:

```
procedure Reduce(D, n) ;
  begin
    for b ← 1 to ⌈log n⌉ do
        for each i : 1 ≤ i ≤ n pardo
          if BITS(D(i), logn − 1, b) = BITS(i, logn − 1, b)
```

图



单元

i	COMP(i)
e_k : (u_k, v_k), c_{uk}, c_{vk} ,	
e_j : (u_j, v_j), $cmin_j$, $cmax_j$	

脉冲

1
- 1 1: (2, 4) 0 0
- 2 2
- 3 3
- 4 4
- 5 5

2
- 1 1
- 2 2: (2, 4), 2, 0
- 3 3
- 4 4
- 5 5

·
·
·

6
- 1 1
- 2 2: (4, 1), 0, 1
- 3 3
- 4 4: (5, 4), 0, 4
- 5 5
- (2, 1), 2, 4; (2, 4), 2, 4

7
- 1 1
- 2 2
- 3 3: (4, 1), 0, 1
- 4 4
- 5 5: (5, 4), 5, 2; (2, 4), 2, 4

8
- 1 1
- 2 2
- 3 3
- 4 2: (4, 1), 2, 1; (2, 4), 2, 4
- 5 5
- (5, 4), 5, 2; (5, 4), 2, 5

·
·
·

10
- 1 1
- 2 2: (2, 4), 2, 4
- 3 3
- 4 2: (5, 4), 2, 5
- 5 2
- (4, 1), 2, 1; (4, 1), 1, 2

11
- 1 1: (2, 4), 2, 4
- 2 2
- 3 3: (5, 4), 2, 5
- 4 2
- 5 1: (4, 1), 1, 2

·
·
·

15
- 1: (4, 1), 1, 2
- 2 1
- 3 3
- 4 1
- 5 1

图 4.4 Savage的连通分支算法

```
                  then RAR(D(i), D(D(i)))
              endif
          endfor
      endfor
  end .
```

详细算法的形式化描述如下:

算法4.6 CONNECTED – COMPONENTS ON MESH ARRAY

输入：每个处理器 i 存放着邻接表adj(i,j)，$1 \leqslant i \leqslant n$，$1 \leqslant j \leqslant d$；

输出：每个处理器 i 有一局部变量$D(i)$，它是顶点 i 所在连通分支标识。$D(i)$等于 i 所在连通分支的最小标号顶点$(1 \leqslant i \leqslant n)$。

```
begin
  (1) for each i : 1≤i≤n pardo   /* 初始化 */
          D(i) ← i
      endfor;
    for b ← 0 to ⌈logn⌉ − 1 do
  (2) for each i : 1≤i≤n pardo
          candidate (i) ← ∞
      endfor;
  (3) for j ← 1 to d do
        for each i : 1≤i≤n pardo   /* 每个顶点找邻接顶点的最小超顶点 */
            RAR (temp(i), D(adj (i)));
            if temp(i) = D(i) then temp(i) ← ∞ endif;
            candidate(i) ← min{candidate(i), temp(i)}
        endfor
      endfor;
  (4) for each i : 1≤i≤n pardo   /* 找每个超顶点的最新超顶点 */
          RAW (candidate(i), D(D(i)));
          if D(i) = ∞ then D(i) ← i endif;
          if D(i) > i then RAR(D(i), D(D(i)) endif
      endfor;
   (5) Reduce(D, n)   /* 用新的标识去标识每个顶点 */
    endfor
end .
```

在给出算法复杂性之前，首先引入一个引理，它在分析复杂性时要用到。

引理4.6 在n个处理器组成的二维网孔上，实现过程RAR及过程RAW需$O(\sqrt{n})$时间。

证明 利用网孔上排序算法即可，详细证明见[13]．

定理 4.8 已知无向图 $G(V,E)$，其中 $|V|=2^k=n$，G 的度数为 d，那么在 n 个处理器组成的二维网孔上求 $G(V,E)$ 的所有连通分支算法 4.6 需 $O(dn\log n)$ 时间．

证明 因为算法 4.6 的第 (3) 步需 $O(dn)$ 时间，而第 (3) 步需执行 $\lceil\log n\rceil$ 次，故整个算法需 $O(dn\log n)$ 时间和 $O(n)$ 处理器。

4.7 小 结

本章主要讨论了求解图论的一个基本问题 — 连通分支的各种并行算法．重点介绍了传递闭包法和顶点倒塌法．其中：基于 SIMD − CREW PRAM 及 SIMD − CRCW PRAM，介绍了一个用传递闭包法求连通分支的算法；基于 SIMD − CREW PRAM，给出了基本的顶点倒塌法及改进后的最优连通分支算法；还介绍了在这种计算模型上，当图以邻接表输入时，顶点倒塌法的实现细节；最后，在一维线性阵列和二维网孔上，给出了顶点倒塌法的实现算法。

在本章的开始，我们就已经叙述过，求图的连通分支是并行图论算法研究的热门课题之一。近些年来，关于这一领域的研究，已经取得了许多成果，在这里我们试图对这些成果做一个简单的概括，见表 4.1，有兴趣的读者可以继续进行这一领域的研究．

在共享存贮模型上，Savage 等人[6] 的结果、是对算法 4.3 作了某些修改后得出的．Nath 等人[15] 通过建立链结构和数据的多个副本，解决了数据的“读冲突”，实现了算法 4.3．Shiloach 等人[16] 和 Vishkin[17] 于 1982 年和 1984 年的结果，巧妙地利用了顶点倒塌方法．Reif 等人[18] 则在模型中引入随机性，使得“写冲突”随机地解决．也就是说，若 i 个处理器同时往某一个共享存贮单元写数据时，则只有一个写成功．究竟是那一个写成功，事先并不知道、完全由生成的随机数来决定．这样，每个处理器写入成功概率是 $1/i$，$1\leqslant i\leqslant n$．他们的概率并行算法分为两个阶段。第一个阶段是 $2\log\log n$ 次循环的随机执行阶段，将顶点倒塌为超顶点的初始化过程，且每次循环仅需 $O(1)$ 时间．在第二阶段则执行 Shiloach 等人的算法．结果得到表 4.1 中列出的期望时间复杂性和处理器复杂性，且执行时间超过 $O(\log\log n)$ 的概率为 $o(1/n)$．Peper[19] 对求解图的连通分支问题，提出一个所谓的“灯塔”算法．Kruskal 等人[30] 研究的并行算法，适用于求稀疏图的连通分支问题，他们为链表计数设计出一个更为有效的算法，利用该算法得到两种共享存贮模型上的并行算法，其结果示于表 4.1 中．Han 等人[31] 细致地考察了顶点倒塌法，发现该法每次循环迭代时，只有超顶点数成几何级数减少，而边未必能达到这一点，因而这种方法应用到稀疏图上求连通分支，效果并不理想．1990 年他们提出了一个在每次循环时既考虑顶点归约又考虑边归约的连通分支并行算法，在 SIMD − CREW PRAM 模型上，算法需 $O(m/p+n\log n/p+\log^2 n)$ 时间和 $O(p)$ 处理器，$1\leqslant p\leqslant m/\log^2 n$．而且他们还进一步提炼了该并行算法，得出表 4.1 中的结果．他们令 $P_\alpha(1,k)$

$=\alpha^{k}$，$P_{\alpha}(i,k)=\alpha^{P_{\alpha}(i-1,k)}$，则有 $h_{\alpha}(k,a)=\min\{i \mid P_{\alpha}(i,k)\geq a\}$，故 $H(m,n,p)=m\cdot h_{2}(m/n,\ \log(n/p\log p))$，它非常接近于 m。

表 4.1　求图的连通分支并行算法

年份	作　者	模　型	时间复杂性	处理器复杂性	备　注
1981	Savage 等人[6]	SIMD–CREW PRAM	$O(\log d\ \log n)$	$O(n^2/\log n)$	d 为图的直径
1982	Nath　等人[15]	SIMD–EREW PRAM	$O(\log^2 n)$	$O(n^2/\log n)$	
1982	Shiloach & Vishkin[16]	SIMD–CRCW PRAM	$O(\log n)$	$O(n+2m)$	
1982	Reif 等人[18]	SIMD–CRCW PRAM	$O(\log\log n)$	$O(n^2)$	期望复杂性
1987	Peper 等人[19]	SIMD–CRCW PRAM	$O(n)$	$O(n)$	
1985	Gopalakrishnan 等人[20]	SIMD–q–维网孔	$O(q^2(m^{1/q}+n^{1/q})\log n)$	$O(m)$	稀疏图
1979	Wyllie[21]	SIMD 树机	$O(\log^2 n)$	$O(n+2m)$	
1984	Yeh 等人[22]	Systolic 树机	$O(n^2/p)$	$O(p)$	$1\leq p\leq n$
1987	Awerbuch 等[23]	SIMD 洗牌网络	$O(\log^2 n)$	$O(n^2)$	
1984	Ullman[24]	SIMD 树网	$O(\log^3 n)$	$O(n^2)$	
1985	Huang[25]	SIMD 树网	$O(n^2/p)$	$O(p)$	$1\leq p\leq \frac{n^2}{\log^2 n}$
1980	Van Scoy[27]	二维细胞阵列	$O(n)$	$O(n^2)$	细胞处理器
1981	Lipton 等人[28]	VLSI 的二叉树机	$O(n^{1+\varepsilon})$	$O(n^{1+\varepsilon})$	$0<\varepsilon<1$
1986	Kruskal 等人[30]	SIMD–EREW PRAM	$O(m/p+n\log p/p+p^{1+\varepsilon})$	$O(p)$	$p\leq m^{\frac{1}{2-\varepsilon}}$，$\log p\leq m/n$，$0<\varepsilon<1$
1986	Kruskal 等人[30]	SIMD–CREW PRAM	$O(\frac{m}{p}+\frac{n\log p}{p}+p\log p)$	$O(p)$	$p\leq\sqrt{m/\log m}$，$\log p\leq m/n$，$0<\varepsilon<1$
1990	Han 等人[31]	SIMD–CREW PRAM	$O(\frac{H(m,n,p)}{p}+\frac{n\log n}{p\log(\frac{n}{p})}+\log^2 n)$	$O(p)$	$1\leq p\leq m/\log^2 n$，$H(m,n,p)$ 非常接近 m

在互连网络模型上，Gopalakrishnan 等人[20] 基于 q－维网孔模型，对求稀疏图的

连通分支，提出了一个并行算法．Wyllie[21] 则基于树机模型应用顶点倒塌方法，给出一个并行算法，使用一个双向循环链表结构存贮每一个顶点的邻接表，其结果包含在表 4.1 中．Yeh 等人[22] 基于 **Systolic** 树机模型，建议了一个连通分支并行算法．Awerbuch 等[22] 在洗牌网络模型上提出了相应的算法．Ullman[24] 则基于树网这种特殊 VLSI 结构，给出一个并行地求图的连通分支算法．随后，Huang[25] 基于同一模型得到一个更为有效的连通分支并行算法。Levitt 等人[26] 和 Van Scoy[27] 分别给出二维细胞阵列上的并行算法，表 4.1 列出了后者的结果．Lipton 等人[28] 和 Hochschild 等人[29] 基于 VLSI 的二义树模型，分别建义了求解图的连通分支并行算法．梁维发和陈国良在超立方上建议了最优的连通分支算法[32]．

参 考 文 献

[1] Hirschberg D S. Parallel Algorithms for the Transilive Closure and the Connected Component Problems, *Conf. Rec. 8th Ann. ACM Symp. on Theory of Comput.*, 1976, 55–77

[2] Hirschberg D S, Chandra A K, Sarwate D V. Computing Connected Components on Parallel Computers, *Commu. ACM*, **22**(8), 1979, 461–464

[3] Chin F Y, Lam J, Chen I N. Optimal Parallel Algorithms for the Connected Component Problem, *Proc.* 1981 *Intern. Conf on parallel Processing*, 1981, 170–175

[4] Chin F Y, Lan J, Chen I N. Efficiennt Parallel Algorithms for Some Graph Problems, *Commu. ACM*, **25**(9), 1982, 659–665

[5] Reghbati E, Corneil D G. Parallel Computations in Graph Theory, *SIAM J. Comput.*, 7(2), 1978, 230–237

[6] Savage C, Ja'Ja' J. Fast Efficient Parallel Algorithms for Some Graph Problems, *SIAM J. Comput.*, **10**(4), 1981, 682–690

[7] Kucera L. Parallel Computation and Conflicts in Memory Access, *Inform. Proc. Lett.*, **14**(2), 1982, 93–96

[8] Kung H T. Why Systolic Architectures?, *Computer*, **15**(1), 1982, 37–46

[9] Savage C. Parallel Algorithms for Graph Theoretic Problems, Ph. D dess. Mathematics Dept., Univ. of Illinois, Urbana, Ill., 1977

[10] Savage C. A Systolic Design for Connectivity Problems, *IEEE Trans. Comput.*, **C–33**(1), 1984, 99–104

[11] Preparata F P. Parallelism in sorting, *Intern. Conf on Parallel Processnig, Bollair, Michigan*, 1977

[12] Nassimi D, Sahni S. Finding Connected Components and Connected Ones on a Mesh–Connected Parallel Computer, *SIAM J. Comput.*, **9**(4), 1980, 744–757

[13] Nassimi D, Sahni S. Data Broadcasting in SIMD Computers, *IEEE Trans. Comput.*, **C–30**(2), 1981, 101–107

[14] 陈国良，并行算法—排序和选择，合肥：中国科学技术大学出版社，1990

[15] Nath D, Maheshwari S N. Parallel Algorithms for the Connected Components and Minimal Spanning Tree Problems, *Inform. Proc. Lett.*, 14(1), 1982, 7–11

[16] Shiloach Y, Vishkin U. An O(log*n*) Parallel Connectivity Algorithm, *J. Algorithms*, **3**, 1982,57–67

[17] Vishkin U. An Optimal Parallel Connectivity Algorithm, *Discrete Applied Math.*, **9**, 1984, 197–207

[18] Reif J H, Spiraris P. The Expected Time Complixity of Parallel Graph and Digraph Algorithms, TR–11–82, Aiken Computation Lab., Harvard Univ., Cambridge, Mass., 1982

[19] Peper F. Determing Connected Components in Linear Time by a Linear Number of Processors, *Inform. Proc. Lett.*, **25**, 1987, 401–406

[20] Gopalakshnan P S, Ramarkishnan I V, Kanal L N. An Efficient Connected Components Algorithm on a Mesh Connected Computer, *Proc.* 1985 *Int'l Conf. on Parallel Processing*, 1985, 711–714

[21] Wyllie J C. The Complexity of Parallel Computations, Ph. D diss., Dept. of Computer Science, Cornell Univ., Ithaca , N Y 1979

[22] Yeh D Y, Lee D T. Graph Algorithms on a Tree–Structured Parallel Computer, *BIT*, **24**, 1984, 333–340

[23] Awerbuch B, Shiloach Y. New Connectivity and MSF Algorithms for Shuffle–Exchange Network and PRAM, *IEEE Trans, Conput.*, C–36, 1987, 1258–1263

[24] Ullman J D. *Computational Aspects of VLSI*, Computer Science Perss, Rocrville, *MD.*, 1984.

[25] Huang M D. Solving Some Graph problems with Optimal or Near–Optimal Speedup on Mesh–of–Trees Network, *Proc. 26th Annu. IEEE FOCS*, 1985, 232–240

[26] Livitt K N, Kautz W T. Cellular Arrays for the Solution of Graph Problems, *Commu. ACM*, 15(9), 1972, 789--801

[27] Van Sccoy F L. The Parallel Recognition of Classes of Graphs, *IEEE Trans. Comput.*, C–**29**(7), 1980, 563—570

[28] Lipton R J, Valdes J. Census Functions : An Apprach to VLSI Upper Bounds, *Proc. 22nd IEEE FOCS*, 1981, 13—22

[29] Hochschild P H, Mayr E W, Siegel A R. Techniques for Solving Graph Problems in Parallel Environments, *24th IEEE FOCS*, 1983, 351—359

[30] Kruskal C P, Rudolpk L, Snir M. Efficient Parallel Algorithms for Graph Problems, *Proc.* 1986 *Intern. Conf. on Parallel Processing*, 1986, 869—876

[31] Han Y, Wagner R A. An Efficient and Fast Parallel Connected Component Algorithm, *J. ACM*, **37**(3), 1990, 626—642

[32] 梁维发，陈国良.多处理器上图的最优算法，计算机学报，**14**(9), 1991

第五章　最小生成树的并行算法

在一个无向连通的加权图 $G(V,E)$ 中，每一条边 (i,j) 都赋以一个权值 $w(i,j)$，从图 G 中找出一棵生成树，若它的权值和为最小，则称这棵生成树为图 G 的一棵最小生成树(Minimum Spanning Tree)，简记作 MST。当图 G 含有权值相等的边时，将会有多棵 MST，为了简化后面的讨论，我们约定 G 中边的权值都不相同。事实上，若有相同的边，则可把三元组 $(w(i,j)$，$\min\{i,j\}$，$\max\{i,j\})$ 的字典序作为赋给每条边的权值，其中：$w(i,j)$ 表示顶点 i 与顶点 j 之间的边 $(i,j)\in E$ 的权值。若图 G 不是连通的，则下面的算法执行结果是由多棵最小生成树组成的森林（最小生成森林 MSF）。

在找 $G(V,E)$ 的 MST 算法中，经典的串行算法有 Sollin 算法[7]，Prim－Dijkstra 算法[8,9] 以及 Kruskal 算法[10]。本章介绍的并行算法，基本上都是上述三种算法的并行化版本。下面各节，我们将介绍找图 G 的 MST 并行算法。

5.1　Sollin算法的并行化

Sollin 算法并行化是从 n 个孤立顶点的森林开始，每个顶点被看作一棵树。在一次循环过程中，算法同时（并行地）对森林中的每一棵树确定该树中任一给定的顶点与某个其它树中顶点连接的最小边。除了两棵树不是用多于一条边连接的这种情况外，所有这些边都被加到森林里。此循环过程不断进行，直至森林中只有一棵树 —— 最小生成树时，算法才终止。因为每次循环时，树的数目至少减少一半，所以 Sollin 算法并行化至多需要 $\lceil \log n \rceil$ 次循环就可找出最小生成树。

在并行环境里，如何标识一棵最小生成树呢？Savage[11] 利用了 Hirschberg 的“顶点倒塌法”，也就是同一棵生成树的顶点都用同一个标识来标识。这样，每棵生成树中最小顶点的标识就成为当前最小生成树的标识。每次找到的一棵生成树的最小边就等于在超顶点之间的最小边。

下面，我们非形式化地描述Sollin算法的并行化：

开始时 $t=0$，令 $G_0=G_0(V,\phi)$，即 G_0 仅含有图 G 的 n 个孤立的顶点。假定对某个正整数 $t>0$ 时已构造成 G_t。若 G_t 是连通的，则 G_t 是图 G 的 MST，算法结束；否则，在 G_t 中找出每个连通分支 C_1 的顶点与另一连通分支 C_2 的一个顶点之间的边，它们是 C_1 与所有其它连通分支的边中权值最小的边，将所有这样的边加入 G_t 中，则形成新的图 G_{t+1}，重复这种过程，最终可求出图 G 的 MST。令 t^* 是 G_t 成为 MST 时 t 的最小值，则 $n/2^{t^*}\geqslant 1$，即 $t^*\leqslant \lceil \log n \rceil$。

我们把从 G_t 到构造完 G_{t+1} 的过程作为一次循环，则上述算法在一次循环时需 $O(\log n)$ 时间和 $O(n^2)$ 处理器。其构造性证明如下:

归纳假设，设 $t \geqslant 0$ 时、G_t 已构造好，图 G 的每个连通分支 C 的最小标号顶点作为 C 的连通分支标识（超顶点的根）．计算函数 $D_t: V \to V$，其中 $D_t(i)$ 表示顶点 i 在第 t 次循环时、i 的连通分支标识。约定 $t=0$ 时，$D_0(i) \leftarrow i$。由 G_t 构造 G_{t+1} 的过程如下:

算法 5.1 PARALLELIZATION OF SOLLIN'S ALGORITHM

输入：无向图G的加权邻接矩阵$W=\{w_{ij}\}$;

输出：G的最小生成树，以逆树形式存贮。

begin

(1) 若$D_t(i)=D_t(j)$，对所有的 $i \in V$，$j \in V$都成立，则G_t是图G的一棵MST，算法终止;

(2) 计算函数 $C: V \to V$。对每个顶点 $i \in V$ 来讲，$C(i)=j$ 当仅当 $W(i,j)$ 是$D_t(i) \neq D_t(j')$的顶点 j'与顶点 i 关联的边中权值最小的边;

(3) 对G_t的每个连通分支C，找出C内的某个顶点 i，使得它的$w(i,C(i))$值最小。将图G中所有这样的边加入G_t中构成为G_{t+1};

(4) 找出G_{t+1}的每个连通分支的连通分支标识。计算函数$D_{t+1}: V \to V$，$D_{t+1}(i)$是 i 所在连通分支的标识

end.

定理 5.1 在 SIMD－CREW PRAM 上，求一无向连通加权图 $G(V,E)$，$|V|=n$ 的 MST，算法 5.1 需 $O(\log^2 n)$ 时间和 $O(n^2)$ 处理器。

证明 首先我们证明，算法 5.1 的每一循环步均需 $O(\log n)$ 时间、$O(n^2)$ 处理器。由 G_t 构造 G_{t+1} 这一步可知（注意 G_t 不连通）：第 (1) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (2) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器，这是因为对每个顶点来讲，至多与 $n-1$ 条边相关联，故计算每个顶点的权值最小关联边需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (3) 步同第 (2) 步类似，故需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (4) 步的实现可用 Hirschberg 方法，故需要 $O(\log n)$ 时间、$O(n^2)$ 处理器。算法至多需执行 $t^* = \lceil \log n \rceil$ 次循环 $(1 \leqslant t \leqslant t^*)$。故整个算法需 $O(\log^2 n)$ 时间和 $O(n^2)$ 处理器。

由此可知，**基于 Sollin 算法的并行化算法，本质上是 Hirschberg 算法的一种简单变形**。因而基于 Hirschberg 算法的全部求连通分支算法都可修改为求 MST 的算法。

定理 5.2 基于 Hirschberg 的顶点倒塌法设计的 MST 算法与基于同一方法的连通分支算法具有相同的时间复杂性及处理器复杂性。

证明 将 Hirschberg 连通分支算法的"每次选取邻接顶点的最小的超顶点"这一步改为"每次选取边的权值是最小的邻接超顶点"，并且把每个连通分支中、所有与其它连通分支的邻接边中的权值最小边记录下来，作为 MST 的边即可。无论是计算时间还是使用的处理器数目，和对应的连通分支算法均有相同的复杂性。

推论 5.1 在 SIMD－CREW PRAM 上，求一无向连通的加权图 $G(V, E)$，$|V|=n$ 的 MST 需 $O(\log^2 n)$ 时间、$O(n^2/\log^2 n)$ 处理器。

证明 由第四章最优连通分支算法可知，求 $G(V,E)$，$|V|=n$ 的所有连通分支需 $O(\log^2 n)$ 时间、$O(n^2/\log^2 n)$ 处理器。又由定理 5.2 可知，基于顶点倒塌方法的 MST 算法和同类的连通分支算法具有相同的时间和处理器复杂性。所以计算图 G 的 MST 需 $O(\log^2 n)$ 时间、$O(n^2/\log^2 n)$ 处理器。

参考文献 [3] 给出了具有推论 5.1 算法复杂性的 MST 算法，为了避免和第四章算法的重复，故在此不再叙述。

5.2 树机上的 MST 算法

Bentley 基于一种心动树机模型（图 1.13），将 Prim－Dijkstra 最小生成树算法移植到树机上，给出了树机上的最小生成树并行算法[2]。

假定图 $G(V,E)$，$V=\{1, 2, \cdots, n\}$。算法的输入是加权的邻接矩阵 W。Prim－Dijkstra 算法的基本思想是：令 $V_1 \subseteq V$ 是最小生成树的顶点集，$T(V_1)$ 是 MST 的边集合。若 $i^* \notin V_1$，$j^* \in V_1$，且 $w(i^*, j^*) = \min\limits_{i \notin V_1, j \in V_1} \{w(i,j)\}$，则

$$V_1 \leftarrow V_1 \cup \{i^*\}, \quad T(V_1 \cup \{i^*\}) \leftarrow T(V_1) \cup \{(i^*, j^*)\}$$

初始化时，不妨置 $V_1 \leftarrow \{1\}$，$T(\{1\}) \leftarrow \phi$；以后每次循环时，假设顶点集 V_1 和边集 $T(V_1)$ 已构造好，对每个 $i \notin V_1$，i 与树的顶点 $j \in V_1$ 关联边的权值为 $w(i,j)$。选取这样一个 j_i，使得 $w(j_i, i) = \min\limits_{i \notin V_1, j \in V_1} \{w(i, j)\}$，记 $l_i = w(i,j_i) = w(j_i,i)$。然后从所有 $i \notin V_1$ 中选取一个 i^*，使得 $l_{i^*} = \min\{l_i \mid i \notin V_1 \text{ 和 } i \in V\}$，把 i^* 加到 V_1 中，边 (i^*, j_{i^*}) 加到 $T(V_1)$ 中，修改剩下的 $i \notin V_1$ 的 j_i 值和 l_i 值，经过 $n-1$ 次循环后，则 $V_1 = V$，这时的 $T(V_1)$ 即含 MST 的所有边。

现在，我们形式化地描述树机上的MST并行算法如下：

算法5.2 MINIMUM SPANNING TREE ON TREE MACHINE

输入：每个叶处理器 i 存贮着 i 的关联边权值向量 $w(i,j)(1 \leqslant j \leqslant n)$；若 i, j 不关联，则置 $w(i,j)=\infty$；

输出：每个叶处理器 i 有两个局部变量集合 $V_1(i)$，和 TV_i，它们分别存贮MST的部分

顶点和部分边．整个MST为$\bigcup_{i=1}^{n} TV_1(i)$．

begin

(1) **for** each i: $1 \leqslant i \leqslant n$ **pardo** /＊初始化＊/

  if $i=1$ /＊第一个处理器工作＊/

   then $V_1(1) \leftarrow \{1\}$; $TV_1(1) \leftarrow \phi$; $l_1 \leftarrow \infty$; $j_1 \leftarrow 1$; $\text{flag}(1) \leftarrow 0$

    /＊flag(1)＝1 表示处理器1是活动的＊/

   else $V_1(i) \leftarrow \phi$; $TV_1(i) \leftarrow \phi$; $l_i \leftarrow w(i,1)$; $j_i \leftarrow 1$; $\text{flag}(i) \leftarrow 1$

  endif

 endfor;

(2) root broadcasts vertex $i^* = 1$ to all leaf processors by tree;

 /＊将i^*广播到所有叶子中＊/

 for $k \leftarrow 1$ **to** $n-1$ **do**

(3) **for** each i: $1 \leqslant i \leqslant n$ **pardo**

 if $\text{flag}(i)=1$ /＊当处理器i是活动时，则工作＊/

  then if $l_i > w(i, i^*)$ **then** $l_i \leftarrow w(i, i^*)$; $j_i \leftarrow i^*$ **endif**;

   send (l_i, i) to its father processor

  endif

 endfor;

(4) **for** each non_leaf processors **pardo**

  /＊(l_L, i_L)及(l_R, i_R)分别为左及右儿子送来的信息＊/

  if it is root

   then if $l_L < l_R$ **then** $i^* \leftarrow i_L$ **else** $i^* \leftarrow i_R$ **endif**;

    broadcasts i^* to each leaf processor by tree

   else if $l_L < l_R$

     then send(l_L, i_L) to its father processor

     else send(l_R, i_R) to its father processor

    endif

  endif

 endfor;

(5) **for** each i: $1 \leqslant i \leqslant n$ **pardo**

  if $\text{flag}(i)=1$ /＊将新的MST边加入到TV_1中去＊/

   then if $i=i^*$

    then $V_1(i) \leftarrow V_1(i) \cup \{i^*\}$; $TV_1(i) \leftarrow TV_1(i) \cup \{(i, j_i)\}$; $\text{flag}(i) \leftarrow 0$

    endif

```
        endif
      endfor
    endfor
  end .
```

这个算法的正确性是显然的，证明从略。

定理 5.3 在树机模型上，找一个连通加权无向图 $G(V,E)$，$|V|=n$ 的 MST，算法 5.2 需 $O(n\log n)$ 时间、$O(n)$ 处理器。

证明 算法 5.2 的第 (1) 步需 $O(1)$ 时间、$O(n)$ 处理器；第 (2) 步需 $O(\log n)$ 时间、$O(n)$ 处理器；第 (3) 步需 $O(1)$ 时间、$O(n)$ 处理器；第 (4) 步需 $O(\log n)$ 时间、$O(n)$ 处理器；第 (5) 步需 $O(1)$ 时间、$O(n)$ 处理器；且第 (3) 步至第 (5) 步执行了 $n-1$ 次，故整个算法需 $O(n\log n)$ 时间、$O(n)$ 处理器。

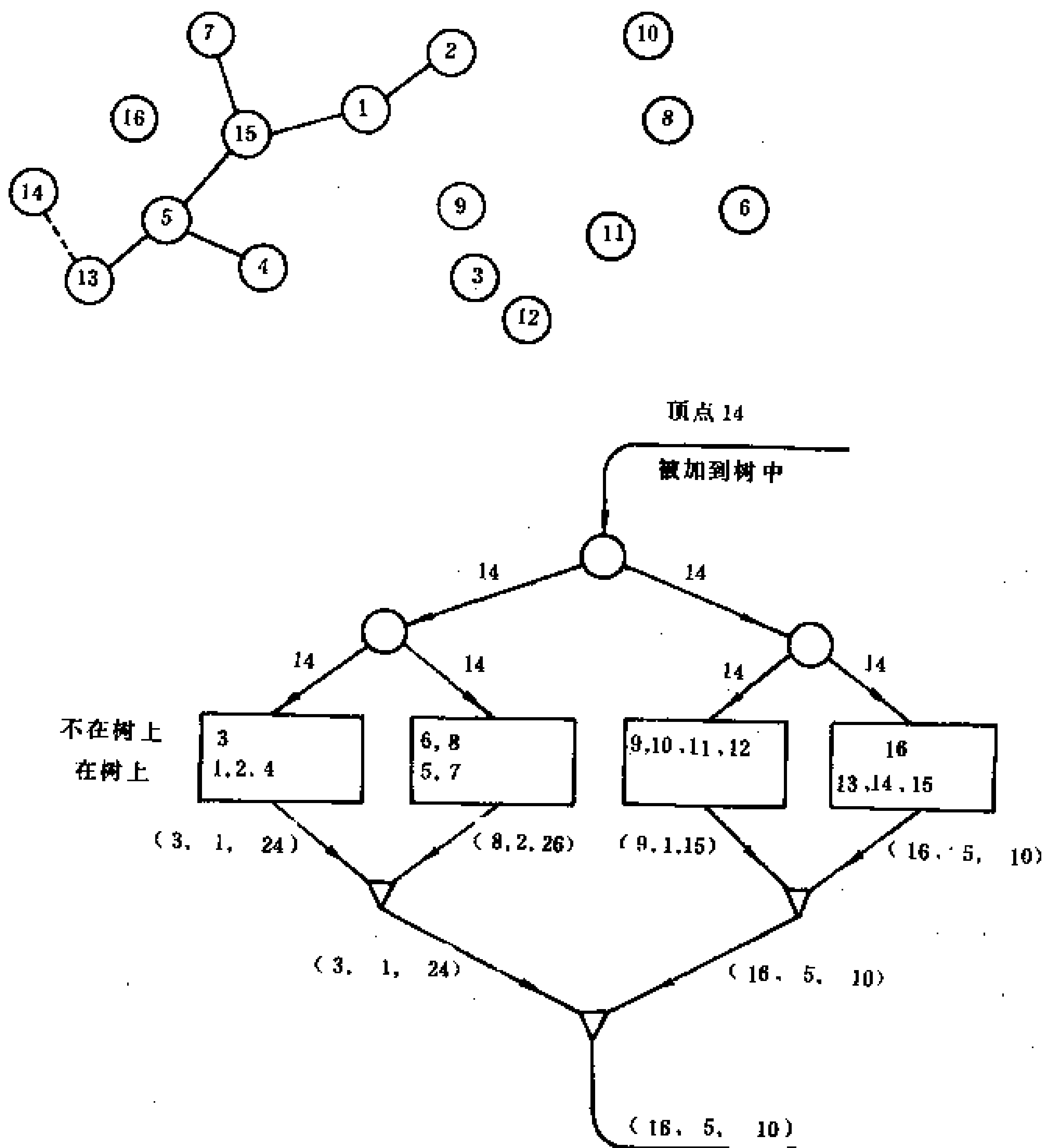


图 5.1 Bentley 的最小生成树算法的一个循环步

每个三元组表示：(1)候选顶点；(2)最邻近的树顶点；(3)与树的距离。

Bentley 还建议了一个改进的算法，在保持时间复杂性不变的前提下，用更少的处理

器树机来计算图的 MST。具体地讲，叶处理器数减少到 $O(n/\log n)$来实现算法，这时，每个叶处理器 $iS(1\leqslant i\leqslant \lceil n/\log n\rceil)$至多含有 $\lceil\log n\rceil$个顶点的加权邻接矩阵的子块 $W((i-1)\lceil\log n\rceil : i\lceil\log n\rceil; 1:n)$，（最后一个叶处理器的邻接矩阵可能不够 $\lceil\log n\rceil$ 个顶点）。位于同一叶处理器内的顶点按顺序计算。由于每个叶处理器至多含 $\lceil\log n\rceil$ 个顶点，故计算时间至多为 $\lceil\log n\rceil$。为此得：

推论 5.2 在 SIMD 树机模型上，求无向连通的加权图的 MST，算法 5.2 使用 $O(n/\log n)$ 处理器可在 $O(n\log n)$ 时间内完成。

证明 类似定理 5.3 的证明。

图5.1例示出用$O(n/\log n)$处理器的树机来实现算法5.2的一个循环步。

5.3 二维网孔上的 MST 算法

5.3.1 算法的基本原理

Atallah 等人[12,13]基于 Sollin 算法和顶点倒塌方法，在二维网孔($n\times n$ 个处理器)上建议了一个 MST 算法。他们的算法的基本思想是：算法由许多次循环组成。在每次循环时，废弃已属于同一个连通分支的边。在两个连通分支之间，若存在两条或两条以上的边，则只保留它们当中权值最小的一条边，删去其它的边。这样图 G 的同一个连通分支内的顶点就形成一个超顶点，下一次循环是在由所有超顶点形成的图上进行操作。

5.3.2 算法的非形式化描述

本节介绍的算法是在 $n\times n$ 个处理器网孔上的并行算法。假定处理器按行主方式编号。算法的输入是加权邻接矩阵，也就是编号为(i,j)的处理器（记作 $PE_{i,j}$）内存放着 $w(i,j)$。算法的输出为 MST 的邻接矩阵 T，$PE_{i,j}$ 内存放着 $T(i,j)$，若 $T(i,j)=1$，则 i 与 j 之间的边是 MST 的边$(1\leqslant i,j\leqslant n)$。

在给出算法之前，首先我们引入在网孔上几种简单的数据移动操作。

1. 水平转动

每个处理器创建一个它的数据副本。然后每行上的数据副本在处理器上象一队士兵一样，首先向左移动，当数据元素到达 $PE_{i,1}$ 时，它弹回继续向右移动，直到它在 $PE_{i,n}$ 处再次被弹回向左移动，当数据元素移动到它开始出发时的位置就停止下来。显然，开始位于 $PE_{i,j}$位置的数据元素访问了第 i 行所有的处理器$(1\leqslant i, j\leqslant n)$。完成这一操作需 $O(n)$时间。

2. 垂直转动

类似于水平转动，初始位于 $PE_{i,j}$ 的数据元素将访问第 j 列所有其它处理器 $(1\leqslant i,j\leqslant n)$，完成这一操作也需 $O(n)$ 时间。

3. 随机存取写

初始时，存在这样一些处理器（处理器数目大于等于 1），它们中的每一个，如 $PE_{i,j}$，含有另一个处理器 $PE_{i',j'}$ 的地址 (i',j')。在一次随机存取写时，我们把 $PE_{i,j}$ 的

内容送入对应的 $PE_{i',j'}$ 中，假定没有两个处理器将它们的数据送入同一处理器中，实现这种操作需 $O(n)$ 时间[14]。

4. **带冲突的随机存取写**

基本上同随机存取写一样，不同的是可能存在多个处理器想把它们的数据写入同一地址，解决这种写冲突的方法是优先权高的处理器写成功（最高优先权可定义为按某种编号方式给处理器编号，如按编号最小的处理器优先权最高）．完成这一操作需 $O(n)$ 时间。

构造MST是由一系列循环组成的，每次循环完成下列计算（初始时 $T=(V,\phi)$）：

(1)　T 的每个连通分支都选择一条连接 T 中两个连通分支的边，且这条边是所有这样的边中权值最小的（假定图 G 的每条边权值不相等）；

(2)　把上一步选择出的那些边加入到 T 中，算法重复地执行 (1) 及 (2)，直到 T 成为一个连通分支时才终止。显然，算法循环的次数不超过 $\lceil \log n \rceil$。

每次循环后，立即废弃掉图 G 中属于 T 的同一个连通分支的边。若两个连通分支之间有多条边连接它们，除保留一条权值最小的边外，废弃掉所有其它多余的边，最后将图 G 的同一个连通分支内的顶点倒塌为一个超顶点。令 G_t 表示第 t 次循环后图 G 倒塌后的归约图 ($G_0=G$)。注意 G_t 的每条边 (i,j) 实际上是图 G 的边 (i',j')。

算法的详细描述如下:

算法5.3　MST ON MESH ARRAY

输入：图G的加权邻接矩阵 $W=\{w_{ij}\}$；

输出：图G的最小生成树T的邻接矩阵 $T=\{T_{ij}\}$。

begin

(1) $G_0 \leftarrow G(V,\ E)$；　$T(V,\ E') \leftarrow (V,\ \phi)$；　$t \leftarrow 1$；

(2) **while** $|G_t|>1$ **do**　/ * $|G_t|$表示G_t的顶点数 * /

(2.1) 对G_{t-1}的每个顶点 v，在G_{t-1}中选择一个与 v 关联且权值最小的边(v,x)，
　　令H_t 是已选出的全部边的集合；

(2.2) 将H_t中的边与G中对应的边放入T中；

(2.3) 构造G_t，将H_t的同一个连通分支内的顶点归并为一个超顶点，使得每两个连
　　通分支之间只保留权值最小的边，同时还废弃属于同一个连通分支内的边

　endwhile

end .

请读者注意，在第 t 次循环执行之前，G_{t-1} 中的顶点数目就是 T 的连通分支个数。因此，若 G_t 有 l 个顶点，则 $l \leqslant n/2^{t-1}$。为了减少在整个网络上数据移动的开销，我们

假定 G_{t-1} 的加权邻接矩阵可以"固定"在一个 $l \times l$ 的处理器子阵列中，因而第 t 次循环时，只需 $O(l)$ 时间。下面我们将会证明，第 t 次循环确实可以在 $O(l)$ 时间内完成。所以整个算法可以在 $\sum_{t=1}^{\lceil \log n \rceil} O(n/2^{t-1}) = O(n)$ 时间内完成。

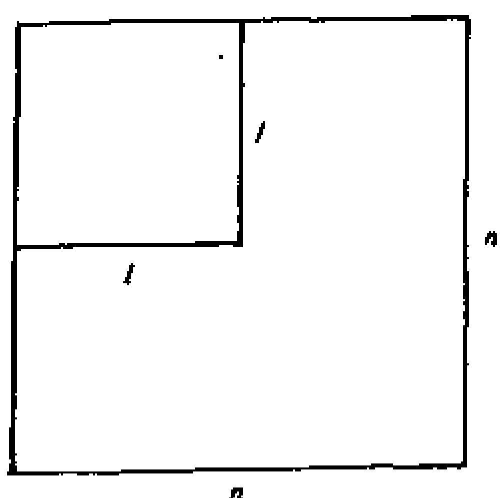


图 5.2 数据在网孔上的存贮示意

在描述第 t 次循环之前，我们简单地描述一下这次循环前、后的邻接矩阵的存放格局。在第 t 次循环开始时，G_{t-1} 的邻接矩阵已存放在左上角的 $l \times l$ 个处理器中(见图 5.2)，若 $G_{t-1}(i,j)=1$，则处理器 $PE_{i,j}$ 含有 G 的边(i',j')。边(i',j')在 G_{t-1} 中表示为(i,j)且边的权值为 $w(i,j)$，即是 $w(i',j')$。此外，在第 t 次循环之前已选出的 T 的边存贮在左上角的 $l \times l$ 个处理器以外的处理器中，每条边作为一个特殊对(x, y)存贮。每个处理器最多可存贮两个这样的特殊对。(在最后一次循环时，邻接矩阵 T 可通过两次随机存取写得到，也就是说含有(x,y)特殊对的 $PE_{i,j}$ 向 $PE_{x,y}$ 发写 $T(x,y)=1$ 的消息）。不难看出，上述的所有要求在第一次循环执行之前是满足的。当第 t 次循环终止时，G_t 的邻接矩阵必须存放在左上角 $l' \times l'$ 个处理器中$(1 \leqslant l' \leqslant l/2)$，而且此次循环产生 T 的边将作为特殊对存贮在左上角的 $l' \times l'$ 个处理器之外，但是在左上角的 $l \times l$ 个处理器之内。

第 t 次循环的初始化是执行一个水平转动（指在左上角 $l \times l$ 个处理器子阵列上）操作，操作结果，每个 $PE_{i,i}$ $(i \leqslant l)$ 在 G_{t-1} 中选择了一条权值最小的边 (i, x)，在水平转动之后，接着执行一个垂直转动，选择了边 (i, x) 的 $PE_{i,i}$ 告诉 $PE_{i,x}$ 及 $PE_{x,i}$ 置 $H_t(i, x) = 1$ 和 $H_t(x, i) = 1$，然后每个 $H_t(i, j) = 1$ 的处理器创建一个特殊对 (i', j')。现在，形成 G_t 的邻接矩阵并把 G_t 嵌入左上角的 $l' \times l'$ 个处理器之前，我们必须清除那些含有特殊对的处理器。又因为 $l' \leqslant l/2$，所以只要清除左上角的 $(l/2) \times (l/2)$ 个处理器即可。完成这一任务只需 $O(l/2)$ 时间。即左上角 $(l/2) \times (l/2)$ 个处理器的特殊对垂直向下移动 $l/2$ 次。在经过 $l/2$ 次移动后，位于 $PE_{x,y}(1 \leqslant x, y \leqslant l/2)$ 内的特殊对移到了处理器 $PE_{x+l/2,y}$ 内 $(x + l/2 \leqslant l)$。

下面创建 G_t 的邻接矩阵。具体过程是：首先，计算矩阵 H_t 的传递闭包 H_t^*，然后作水平转动。这时每个 $PE_{i,i}$ $(1 \leqslant i \leqslant l)$ 计算出 $D(i)$ —— i 所在连通分支标识（即此连通分支的最小标号顶点）；然后作水平转动和垂直转动，将 $D(i)$ 播送到第 i 行及第 j 列；再次，每个 $PE_{i,j}$ 做：若 $G_{t-1}(i,j) = 1$ 且 $D(i) \neq D(j)$，则 $PE_{i,j}$ 创建一个随机存取写 $G_t(D(i), D(j)) \leftarrow 1$ 和 $w_t(D(i), D(j)) \leftarrow w_{t-1}(i,j)$ 消息给 $PE_{D(i), D(j)}$。若写"冲突"，则仅有 $(w(i',j'), i', j')$ 最小的写成功。接下来再做的是：怎样把 G_t 的邻接矩阵压缩到左上角的 $l' \times l'$ 个处理器中去，其方法如下：

(1) $r(i) = |\{k \mid D(k) = k \text{ 和 } k \leqslant i\}|$;

(2) 把 $r(i)$ 送入第 i 行和第 i 列;

(3) $G_t(i,j)=1$的$PE_{i,j}$创建消息告诉$PE_{r(i),r(j)}$置$G_t(r(i),r(j))\leftarrow 1$，$w(r(i),r(j))\leftarrow w(i,j)$且$G_t$的边$(r(i),r(j))$表示$G$的边$(i',j')$，在$PE_{i,j}$创建这一消息后，废弃$(i,j)\in G_t$表示$(i',j')$消息，然后执行随机存取写，则所有消息都送到了目的地。其结果，G_t存贮在左上角的$l'\times l'$个处理器阵列中。

算法执行了 $\lceil \log n \rceil$ 次循环后，每个含有特殊对 (x,y) 的 $PE_{i,j}$ 创建消息 $T(x,y)=1$ 给 $PE_{x,y}$，然后执行随机存取写。又因为每个 $PE_{i,j}$ 至多含有两个特殊对，所以至多两次随机存取写即可。

5.3.3 算法的复杂性分析

在分析算法的复杂性之前，我们先引用将在第六章要证明的引理。

引理 5.1 在 $n\times n$ 的二维网孔上，求 n 阶布尔矩阵 B 的传递闭包需 $O(n)$ 时间。

证明 见本书第六章定理 6.4。

定理 5.4 在 SIMD 的二维网孔上，求无向连通的加权图 $G(V,E)$，$|V|=n$ 的 MST，算法 5.3 需 $O(n)$ 时间、$O(n^2)$ 处理器。

证明 算法 5.3 是由 $\lceil \log n \rceil$ 次循环组成的。第 t 次循环需 $O(l_t)$ 时间 $(l_t\leqslant n/2^{t-1})$。故所有的循环需要的时间为：

$$\sum_{t=1}^{\lceil \log n \rceil} O(l_t)\leqslant \sum_{t=1}^{\lceil \log n \rceil} O(n/2^{t-1})=O(n)$$

而最后的随机存取写需 $O(n)$ 时间。因此整个算法需 $O(n)$ 时间和 $O(n^2)$ 处理器。

5.4 MST 的更新算法

增值图论算法 (Incremental Graph Algorithm) 是指对图作一些增值改变后，重新计算图原先具有的性质的一类算法。所谓**增值**指的是图中顶点的增加或减少，边权值的改变等。本节我们介绍一个 MST 更新算法。

5.4.1 基本概念

我们用(a,b)表示顶点 a 与顶点 b 之间的一条无向边。$\langle a,b\rangle$ 表示一条有向边，其方向是从 a 到 b。约定顶点 u 到顶点 v 的无向路径为$[u-v]$，有向路径为$[u\rightarrow v]$。逆树是一棵有向树，它有一个特殊的顶点，称为树根。若 u 在 v 到根的路径上，则顶点 u 称为顶点 v 的祖先，而 v 称为 u 的一个子孙。一个顶点的父顶点是它的最亲近的祖先。树 T 中顶点 x，y 的**最低公共祖先**(The Lowest Common Ancestor) LCA(x,y)是 x，y 的最亲近的共同祖先，且 x 和 y 的所有其它公共祖先也是 LCA(x,y)的公共祖先。若一棵树的边都按从儿子到父顶点定向，得到的有向树称为**逆树**(Inverted Tree)[17]。逆树中边$\langle a,b\rangle$表示 b 是 a 的父顶

点。一个顶点 v 到根的距离称为 v 的深度。

令 $T=(V',E')$ 是一棵逆树，$V'=\{1, 2, \cdots, n\}$，r 是 T 的根。在根 r 处带有自环的逆树可以用一个函数 $F:V'\to V'$ 来表示，其中：$F(i)$ 是顶点 i 的父顶点 $(i\neq r)$，且 $F(r)=r$。根据这个函数 F，下面我们定义函数 F^k: $V'\to V'$ $(k\geqslant 0)$:

$$F^k(i)=\begin{cases} i, & k=0 \\ F(F^{k-1}(i)), & k\neq 0 \end{cases} \qquad \text{任意} i\in V'$$

现在我们给出计算所有$F^k(i)$ $(0\leqslant k<n)$的算法。假定处理器以行主方式编号，且使用 n^2 个处理器。

算法5.4 ANCESTORS IN INVERTED TREE

输入：逆树T用一维数组 F 表示，其中：$F(i)$表示顶点 i 的父亲。若 i 是树根，则 $F(i)=i$；

输出：每个顶点$i\in V$的所有祖先$F^k(i)$，$0\leqslant k<n$，$i\in V$。

begin

(1) **for** each i: $1\leqslant i\leqslant n$ **pardo** / * PE(i,1)执行指令 * /

$F^0(i)\leftarrow i$;　$F^1(i)\leftarrow F(i)$

endfor;

(2) **for** $t\leftarrow 0$ **to** $\lceil\log(n-1)\rceil-1$ **do**

for each i,s: $(1\leqslant i\leqslant n)\wedge(1\leqslant s\leqslant 2^t)$ **pardo**

$F^{2^t+s}\leftarrow F^{2^t}(F^s(i))$

endfor

endfor

end.

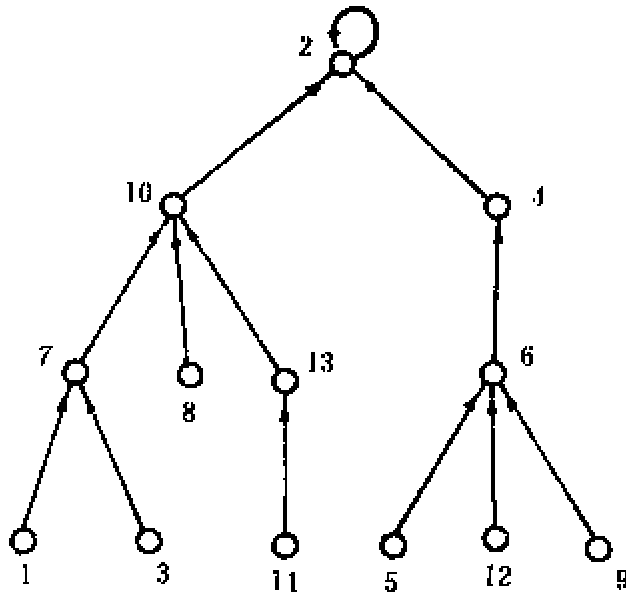
引理 5.2　在 SIMD－CREW PRAM 上，若已知逆树函数 F，则计算所有的 $F^k(i)$，$i\in V$，$0\leqslant k<n$ 的算法 5.4，可使用 $O(n^2)$ 处理器在 $O(\log n)$ 时间内完成。

证明　由算法 5.4 可知，F^k 是由前面的循环计算出来的 F^i，$(i<k)$ 复合而成的。每一次循环，都使顶点的祖先个数增加一倍，而一个顶点至多有 $n-1$ 个祖先，故 $\lceil\log(n-1)\rceil-1$ 次循环就可以计算出全部祖先。在第 t 次循环时需 $O(2^t n)$ 处理器。故整个算法需 $O(\log n)$ 时间、$O(n^2)$ 处理器。

$F^k(i)$，$(1\leqslant i\leqslant n, 1\leqslant k<n)$的实际计算是通过一个二维数组 F^+完成的。$F^+(i,k)$即为$F^k(i)$。一旦F^+计算完成，顶点i $(1\leqslant i\leqslant n)$在树 T 中的深度 depth(i)可以通过对F^+的第 i 行执行二分法搜索来实现，根 r 最左出现的位置（列号）即是 i 的深度。故完成 depth(i)，$(1\leqslant i\leqslant n)$的计算需$O(\log n)$ 时间和 $O(n)$处理器。然后将深度信息存入一维数组 D^+中。在计算出 D^+之后，F^+的每行向右移动，直到最左出现的 r 在最后的一列为止。如图 5.3 所示。

引理 5.3 在 SIMD－CREW PRAM 上，计算一棵逆树 $T(V',E')$，$|V'|=n$ 的所有顶点对 (x,y) 的最低公共祖先需 $O(\log n)$ 时间、$O(n^2)$ 处理器 $(1 \leqslant x,y \leqslant n)$。

证明 因为图 $T(V',E')$有$C_2^n = O(n^2)$个顶点对。令 u，v 为一对顶点，令 w 是它们的最低公共祖先 LCA(u,v)。那么可用一个处理器在F^+ 的第 u 行与第 v 行执行二分法搜索，并且定位这两行最先出现的共同顶点，即是最低公共祖先。故总共需$O(\log n)$时间、$O(n^2)$处理器。



(a) 逆树

F^+	0	1	2	3	4	5	6	7	8	9	10	11	12
1										1	7	10	2
2													2
3										3	7	10	2
4												4	2
5										5	6	4	2
6											6	4	2
7											7	10	2
8											8	10	2
9										9	6	4	2
10												10	2
11										11	13	10	2
12										12	6	4	2
13											13	10	2

(b) 二维数组 F^+

图 5.3 用二维数组计算 F^k

说明：在 F^+行向右平移后，最右列表示树根，空白项表示未定义项。

下面我们讨论通过它们最低公共祖先的任意两个顶点之间的路径 $[u-\text{LCA}(u,v)-v]$ 上权值最大边的计算问题。

令函数 $\text{MAX}_e(e_1, e_2)$ 的返回值是 e_1 和 e_2 中的一个权值较大的边。令 $E_m^k(i)$，$(1 \leqslant i \leqslant n)$ 是 i 到它在 T 中的第 k 辈祖先那条路径上的权值最大边。那么：

(1) $E_m^1(i)$ 是边$(i, F(i))$；

(2) $E_m^k(i)$ 是边$e = \text{MAX}_e(E_m^{k-1}(i), (F^{k-1}(i), F^k(i)))$，$k > 1$；

(3) 假定根r 的边(r, r)为 $-\infty$。

引理 5.4 在 SIMD－CREW PRAM 上，所有的 $E_m^k(i)$，$1 \leqslant i \leqslant n$，$1 \leqslant k < n$ 的计算需 $O(\log n)$ 时间、$O(n^2)$ 处理器。

证明 类似于引理 5.2 的证明。$E_m^k(i)$ 的计算可以用二维数组 E^+ 来完成。$E^+(i,k)$ 是从顶点 i 到它的第 k 辈祖先这条路径上权值最大的边。

引理 5.5 在 SIMD－CREW PRAM 上，已知逆树 $T(V',E')$ 的二维数组 F^+，E^+ 和一维数组 D^+，计算所有 $u \in V'$，$v \in V'$ 之间的路径 $[u-v]$ 上权值最大的边需 $O(\log n)$ 时间

和 $O(n^2)$ 处理器。

证明 令 w 是顶点 u，v 的最低公共祖先 LCA(u,v)。同时令 w 是 u 的第 x 辈祖先，w 是 v 的第 y 辈祖先，x 和 y 的值可以从深度信息得到。那么路径 $[u-v]$ 上权值最大的边是 $\mathrm{MAX}_e(E_m^x(u),\ E_m^y(v))$。

5.4.2 顶点更新的 MST 算法

当从原来图中删除或插入一个顶点时，都涉及到重新构造新的 MST 问题。本节我们仅考虑顶点插入问题。因为从图中删除顶点，重新构造 MST 困难较大。例如，若原来的 MST 是星形结构，当删去中心顶点后，则将删除掉所有的树边。

1. 算法的基本原理

Pawagi 等人[15,16] 给出的算法，其关键在于怎样利用不太多的处理器，快速地检查旧的 MST 的边以及由插入顶点引进的新边。我们注意到：与新顶点关联的一对边将导致在旧的 MST 中形成一个环。而与新顶点关联边的数目至多为 n，故在旧的 MST 中创建了 $C_2^n = O(n^2)$ 个环。算法的基本思想是：移去环中权值最大的边，将所有的环断开，结果得到的图就是更新后的 MST。

2. 算法的非形式化描述

算法5.5 UPDATING MST BY ADDING A NEW VERTEX

输入：原来的最小生成树（以逆树形式存贮）和更新后新图的加权邻接矩阵 W；

输出：新的 MST（仍以逆树形式存贮）。

begin

/ * 令 z 是插入到图中的新顶点 * /

(1) 计算旧的 MST 的 M^+； / * $M^+ = \mathrm{MAX}_e$ * /

(2) 计算旧所有由 z 导出的环 C 中权值最大边。例如，令 u，v 是 MST 的两个顶点，且令 (z,u)，(z,v) 是同 u，v 关联的两条新的边。由 (z,u) 与 $[u-v]$ 及 (v,z) 形成的环中权值最大的边是 (z,u)、(z,v) 及 $M^+(u,v)$ 中的一个。若存在权值相等的边，则按$(w(u,v),\ \min\{u,v\},\ \max\{u,v\})$字典序选取；

(3) 删除第(2)步选出的边。由于可能存在多个处理器选择了同一条边，这样删除时引起写冲突，可采用文献[18]的方法解决之；

(4) 将新的MST仍以逆树形式存贮。具体方法是：在删除旧的MST的边时创建了若干子树，现在用与z的关联边将这些子树连接成一棵新的MST。设创建了k棵子树，其根分别为x_1，x_2，…，x_k，它们与z关联的顶点分别为w_1，w_2，…，w_k。计算删除选择边后旧的 MST 的 F^+，F^+ 这个数组含有 w_1到 x_1，…，w_k 到 x_k 的路径。将这些路径上边的方向反过来，将(w_1,z)，…，(w_k,z)定向，其方

向指向 z。这样 z 就成了新的 MST 的根

end.

定理 5.5 算法 5.5 是正确的.

证明 令 G_z 是由新的顶点 z 加入旧的 MST 中形成的图. 在新顶点 z 插入原来 MST 时引起 MST 更新问题。令 T' 是上述顶点更新算法执行后得到的图。首先 T' 是无环图。因为在第 (3) 步，我们将所有的环都断开了. 现在假定 T' 是由几个连通分支组成的. 考虑下列的边集合 E_d:

$$E_d = \{e \mid e\text{是}T'\text{的两个连通分支之间的边、且}e\text{ 是 }G_z\text{的删除边}\}.$$

令 $e_{\min}$ 是集合 E_d 的权值最小边。我们考虑 G_z 中含有 $e_{\min}$ 边的环 C。环 C 至少有 T' 的两个连通分支的顶点. 因此，E_d 中至少有两条边. 不难看出，$e_{\min}$ 不可能被删除，这是因为它不是 C 上权值最大的边。因此 T' 只有一个连通分支。又因为每条非树边是形成环的权值最大边，而算法 5.5 恰恰删除了每个环上的权值最大边. 所以 T' 是 MST.

定理 5.6 在 SIMD－CREW PRAM 上，通过在图 $G(V,E)$，$|V|=n$ 中插入一个顶点引起的 MST 更新，算法 5.5 需 $O(\log n)$ 时间、$O(n^2)$ 处理器.

证明 由引理 5.5 可知，完成算法 5.5 的第 (1) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；由该引理还知道. 完成算法的第 (2) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；而第 (3) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器[18]；第 (4) 步计算 F^+ 需 $O(\log n)$ 时间、$O(n^2)$ 处理器，从 w_1 到 x_1，…，从 w_k 到 x_k 的路径上将方向进行反向，需 $O(1)$ 时间和 $O(n^2)$ 处理器. 所以顶点更新算法 5.5 需 $O(\log n)$ 时间、$O(n^2)$ 处理器。

5.4.3 边更新的 MST 算法

当原图的一条边的权值发生改变时，需重新构造 MST。若树边的权值减少或非树边权值增加，那么旧的 MST 将不作任何改变；反之，若树边权值增加或非树边权值减少，那么旧的 MST 可能发生改变。然而，在上述的两种情况下，至多有一条边成为树边和一条树边成为非树边。

1. 算法的基本原理

(1) 若一条树边(x,y)的权值增加，则MST修改如下:

1° 删除树边(x,y)，这时创建了两棵子树组成的森林;

2° 对每棵子树赋予其连通分支标识;

3° 找出连接这两个连通分支且权值最小的一条边。

(2) 若非树边(u,v)的权值下降，则MST修改如下:

1° 把(u,v)这条边加入到旧的MST中;

2° 移去(u,v)所在的环上权值最大的边.

2. 算法的非形式化描述

算法5.6 UPDATING MST BY CHANGING EDGE'S WEIGHT

输入：以逆树形式存贮的旧的MST以及更新后新图的加权邻接矩阵W；

输出：用逆树形式存贮更新后的MST。

begin

(1) 若权值改变的边(x,y)是非树边，则转到第(8)步；否则执行下列各步：

(2) $F'(x)\leftarrow x$，这样，从旧的MST中删除了边(x,y)，创建了两棵树组成的森林，它们的根分别为r和x；

(3) 计算 F^+，每个顶点的连通分支用根结点标识（从 F^+ 最右列可以看出）；

(4) 对每个顶点 i 找出它与另一连通分支连接的权值最小边 $l_i(1\leqslant i\leqslant n)$，然后从$\{l_i\}$中选出权值最小边 (u,v)；

(5) 若 $(u,v)=(x,y)$ 或 $(u,v)=(y,x)$，则 $F'(x)\leftarrow y$，也就是说，旧的 MST 不作改变；否则将 (u,v) 加入到森林中去；

(6) 新的MST仍旧用逆树形式存贮。令u的根为x，置 $F(u)\leftarrow v$，这样，就给边 (u,v) 定了方向．在 F^+ 中已找到了路径 $[u\rightarrow x]$，现在将 $[u\rightarrow x]$ 上的边都反向，结果得到了 $[x\rightarrow u]$；

(7) **stop**。

(8) 计算 F^+，E^+ 和 D^+，找出 $[x-y]$ 路径上权值最大的边 (u,v)；

(9) 若边 (u,v) 的权值小于边 (x,y) 的权值，则旧的MST不变；否则删除边 (u,v)，同时把边 (x,y) 加入删除后的MST中；

(10) 保持MST仍为逆树；置 $F(x)\leftarrow y$；对 F^+ 中的第u行 $[x\rightarrow u]$ 的路径上边反向（假定u与x是祖先—子孙关系）

end．

定理 5.7 在 SIMD－CREW PRAM 上，图中一条边权值的改变所引起的 MST 更新，算法 5.6 需 $O(\log n)$ 时间、$O(n^2)$ 处理器．

证明 首先我们分析算法 5.6 的各步复杂性，然后再得出整个算法的复杂性．算法的第 (1)、(2) 步只需 $O(1)$ 时间、$O(1)$ 处理器；由引理 5.2，第 (3) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (4) 步同第四章 Hirschberg 算法第 (3) 步有相同复杂性，即需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (5) 步只需 $O(1)$ 时间及处理器；第 (6) 步只需 $O(1)$ 时间、$O(n)$ 处理器；第 (8) 步由引理 5.2 至引理 5.5 可知。完成这一步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (9) 步只需 $O(1)$ 时间及处理器．第 (10) 步同第 (6) 步有相同的时间及处理器复杂性．所以，整个算法需 $O(\log n)$ 时间、$O(n^2)$ 处理器。

5.5 MIMD 共享存贮模型上的 MST 算法

Quinn[19] 基于 MIMD 紧耦合共享存贮模型，实现了 Sollin 算法的并行化。因为他的算法完全基于顺序的 Sollin 算法，所以我们首先给出单机的 Sollin 算法。

算法5.7 SOLLIN'S ALGORITHM (SISD)

输入：无向图$G(V,E)$的加权邻接矩阵$W=\{w_{ij}\}$，$i, j\in V$；

输出：G的最小生成树T（树以边的形式存贮）.

```
begin
(1) for i←1 to n do   /＊初始化＊/
        vertex i is initially in set i
    endfor;
(2) T←φ;   /＊T是MST＊/
(3) while |T|<n－1 do   /＊当T的边数小于 n－1 执行循环＊/
(4)     for each tree i do   /＊每棵树 i 找不在同一树上的最小权值边＊/
            closest(i)←∞
        endfor;
(5)     for each (v,u)∈E do
            if FIND (v)≠FIND(u)
               /＊若为真，说明 v 与 u 属于不同的连通分支＊/
              then if w(v,u)<closest(FIND(v))
                      then closest (FIND(v))←w(v,u);
                           edge (FIND(v))←(v,u)
                   endif
            endif
        endfor;
(6)     for each tree i do
            (v,u)←edge(i);
(7)         if  FIND(v)≠FIND(u)
               then T←T∪{(v,u)};   /＊将新的树边加入树中＊/
(8)                 UNION(v,u)       /＊归并为一个较大的树＊/
            endif
        endfor
  endwhile
end.
```

此算法使用了两个函数 FIND，UNION，FIND(v) 是找出 v 所在的树的标识；而 UNION(v,u) 是将 v 所在子树 T_1 与 u 所在子树 T_2 通过边 (v,u) 连接成一棵较大的子树且以一个较小子树标识作为归并后的标识，不熟悉的读者可查阅文献[1]。

在紧耦合共享存贮的多机系统上，上述算法并行化如下：

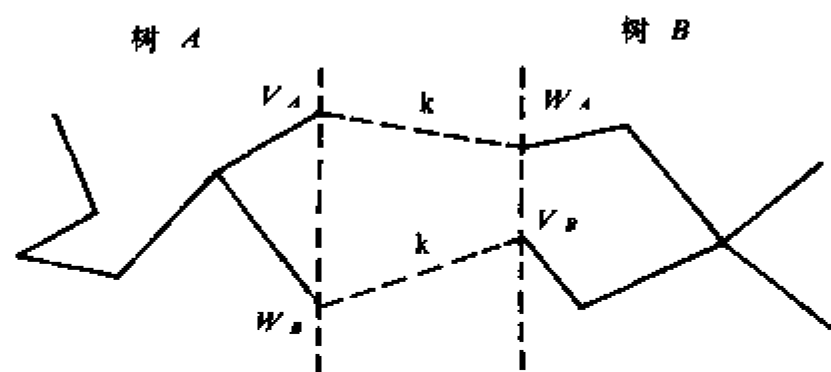


图 5.4 算法 5.7 并行化引起的一种复杂情况

首先我们想到的对 while 循环进行并行化，但遗憾的是：因为循环之间存在先后制约关系，即在第 $k+1$ 次循环执行之前，第 k 次循环时的子树必须同一个有权值最小边关联的另一子树归并，所以 while 循环并行化效率提高是有限的。因此我们在循环体内考虑并行化。算法 5.7 的第(4)步通过适当的伪调度，可以使其并行化。设第 t 次循环时已有 n_t 棵子树。若 $n_t > p$，则把这 n_t 棵子树较均匀地分配到 p 个处理机中，每个处理机约有 $\lceil n_t / p \rceil$ 棵子树；否则，这 n_t 棵子树就分配给前面 n_t 个处理机即可。算法 5.7 的第(5)步并行化，最有效的做法是：首先每一个处理机检查它内部的顶点的边，然后再检查不在同一个处理机上树之间的边；算法 5.7 的第(6)到(8)步并行化则更为复杂。图 5.4 示出了这种情形。假定一个处理机企图将 B 树最近邻居 A 树归并，变量 edge(A)有一条权值为 k 的边(v_A,u_A)，变量 edge(B)也有一条权值为 k 的边(v_B,u_B)，进一步假定两个处理机都在执行 UNION 之前执行了第(7)步的测试，那么(v_A,u_A)以及(v_B, u_B)都加入到树 T 中，结果形成一个环，显然这是错误的。因此，要使第(6)到(8)步并行化，必须在执行第(7)步之前上锁，执行完第(8)步后再开锁。每次仅允许一个处理机进入临界区。为了避免死锁的产生，当有多个请求上锁的进程申请临界区时，仅仅让标号最小的子树上锁。

定理 5.8 在 MIMD— 紧耦合共享存贮模型上，求一无向连通的加权图 $G(V,E)$ 的 MST，算法需 $O(\lceil \log n \rceil (n^2 / p + n / p + n + p))$ 时间和 $O(p)$ 处理机。

证明 为简化起见，假定 FIND、UNION 操作仅需 $O(1)$ 时间；算法的第 (4) 步并行化需 $(n / p + p)$ 时间；第 (5) 步需 $O(n^2 / p + p)$ 时间；第 (6) 至 (8) 步需 $O((n / p)p + p)$ 时间，p 因子是锁步等待时间，整个 while 循环了 $\lceil \log n \rceil$ 次，故整个算法时间为

$$O((n^2 / p + n / p + p + n) \lceil \log n \rceil)$$

处理机数目为 $O(p)$。尤其是当 $p = O(\sqrt{n})$ 时，算法的时间复杂性最小。

5.6 分布式 MST 算法

在通信网络中，从某个结点把一个消息广播到整个网络中，有两种常见的方法：一种称之为**洪水淹没** (Flooding) 法。该法的特点是：网络中的每个结点，一旦收到消息后，立即向所有非父结点广播接收到的消息。这里，网络中的结点第一次从别的结点接收到消

息时，则称那个发送消息的结点为父结点。显然，这种方法的通信开销非常大。另一种方法是**基于网络的一棵生成树广播**，它仅需网络中结点数目减去一的消息个数就够了．因为任意两个结点之间有线路相连接，而通信线路的费用是不一样的，所以我们给每条通信线赋予一个非负的权值．为了达到有效的广播，必须在网络中建立一棵最小生成树 (MST)。因此，在基于异步通信的分布式计算模型上，计算网络的 MST 是一个极其重要而又非常基本的问题。

5.6.1 算法的基本原理

Gallager 等人[3] 建议的 MST 分布式算法，是 Sollin 算法的并行化．他们的算法用到一个非常重要的概念 — **超结点** (Supernode)。超结点有时又称**碎块** (Fragment)。在整个通讯网络中，一个超结点是它的一个子网络。若两个超结点内的结点之间有边连接，则称这个超结点为**邻接超结点**。假定超结点代表的子图的最小生成树已经生成，并且有树根。分布式算法的基本思想是：每个超结点试图选择一个且仅选择一个合适的邻接超结点进行归并，结果得到一个更大的超结点。相应地，把这两个超结点的 MST 归并成一棵更大的 MST。所谓归并，就是在两个超结点之间的边连接中选取权值最小的边加人 MST。反复应用上述的过程，最终得到一个包含整个网络中所有结点的超结点，这个超结点的 MST 就是我们所要计算的 MST。

为了减少同一个结点参与归并过程的次数，我们约定：一个超结点只允许同含有足够多结点的超结点进行归并．要做到这一点，每个超结点 S 都要维持一个层号 level_number(S)，用它来估计一个超结点 S 内所含结点的数目．一个层号为 k 的超结点至少含 2^k 个结点 ($0 \leqslant k \leqslant \lceil \log n \rceil$)。

当一个超结点想同另一个邻接超结点归并时，按照它们的相对层号采取不同的动作．具体地讲，令 S 和 S' 是两个正在考虑的超结点，当 S' 想与 S 归并时，执行动作如下：

(1) 若 level_number(S) > level_number(S')，则 S' 立即与 S 归并，且 level_number(S) 不变，但 S' 内结点的层号改变成 level_number(S)。

(2) 若 level_number(S) < level_number(S')，则 S' 必须等到 S 的层号 level_number(S) 变得足够大时，才能与 S' 进行归并；

(3) 若 level_number(S) = level_number(S')，则它们立即进行归并，并将结果的超结点 S'' 层号改变，即将原 S 和 S' 内结点的层号加 1，即

$$\text{level_number}(S'') = \text{level_number}(S) + 1$$

为了使得在归并过程中所使用的消息数目尽可能地少，每个超结点都要维护它的子网的 MST，且在每个超结点内部，仅通过树边传递消息。

5.6.2 算法的非形式化描述

算法5.8 MST DISTRIBUTED ALGORITHM

输入：每个结点 $v \in V$ 的邻居集 $N(v) = \{u \mid (v,u) \in E\}$ 以及相应的关联边的权值 $w(v,u)$，

```
    ∀u∈N(v);
输出：最小生成树(MST).
begin
(1) for each node v do   /*初始化，标识成超顶点，置标识号*/
        S ← S' ← v;  ID(v) ← v;
        level_number(S) ← level_number(S') ← 0
    endfor;
(2) while there exist more than one supernode do
   (2.1)  each supernode S' tries to find a appropriate neighboring supernode S;
   (2.2)  if level_number(S) < level_number(S')
                     then wait
                     else S'' ← S∪S' ; /*归并成一个超结点*/
                          symbolize edge(u,v) into edge of tree;   /*u∈S, v∈S'*/
                          if level_number(S) > level_number(S')
                                 then ID(S'') ← ID(S);
                                      level_number(S'') ← level_number(S)
                                 else ID(S'') ← min{ID(S), ID(S')};
                                      level_number(S'') ← level_number(S) + 1
                       end if;
                       maintains MST of S''
           end if
    end while
end.
```

一般来讲，一个超结点在修改其MST之后开始进入一个新的阶段。在新的阶段试图找出一个适当的邻接超结点进行归并（即算法5.8的第(2.2)步）。具体做法是：令每个超结点S的MST是T_S，在T_S的根初始化这个新阶段，根结点通过树边向S内所有的结点广播一个“请求”消息M，M要求每个结点$v\in S$去找出邻接超结点中的结点$u\in S'$，使得边(v,u)的权值为最小。一旦找出后，v记住边(v,u)、权值$w(v,u)$和超结点的标识ID(S')；每个结点$v\in S$收到消息M后，发送一个“检测”消息到每条“未检测”的边，该条检测消息含有超结点标识ID(S)，且ID(S)可以是树根结点的编号。在算法初始化时，必须标识所有的边都是“未检测”边；若结点v最终收到来自同一个超结点中结点发来的消息，则用“拒绝”消息应答，且将传递消息那条边的两个端结点分别标识该条边是“无用”边，今后不再对它发送消息；当v最终找到一个满足条件的邻接超结点时（若不存在这样的超结点，则v简单地形成一个“计算终止”消息），若v是T_S的树叶结点，则它将最小权值的边、权值和邻接超结点标识等信息包含在消息内向父结点报告；一个内部结点一直等到收到所有儿子的消息后，才从中找出一条权值最小的边，并与自

已找出的最小权值的边相比较，然后从两者中选出最小权值的边，并形成一个消息向父结点报告，所形成的消息包含最小权值边、权值和邻接超结点等信息内容；最后根结点将决定与哪一个邻接超结点进行归并，它沿着已经建立起来的、由树边组成的路径发送要求“归并”消息给邻接超结点，这条消息通过邻接边到达邻接超结点的某个树结点中，然后再传送到邻接超结点的树根结点上，最后根据这两个超结点的层号，采用相应的策略进行归并处理。

要使两个超结点归并成一个更大的超结点，对树的修改工作甚为简单。将连接这两个超结点的最小权值边标记为树边。令 (v,u) 是这样一条边，且在本阶段开始时 v 与 u 分别隶属于超结点 S' 及 S。仅当 level_number(S') $\leqslant$ level_number(S) 才发生归并。若 level_number(S') $<$ level_number(S)，则 v 结点给 $T_{S'}$ 的根发送一个含有 level_number(S) 和 ID(S)；然后根结点广播更改 $T_{S'}$ 的所有结点的层号和超结点标识，分别改成 level_number(S) 和 ID(S)；再后从 v 到 T_S 的根结点这条路径上的结点，其父子关系重新标识，即父结点现在变成儿子结点，而儿子结点变成为父结点，且 v 在结果的超结点的树中，其父结点为 u。若是 level_number(S) = level_number(S')，则超结点 S'' 的标识 ID(S'') = **min**{ID(S'), ID(S)}；结点 u 和 v 将新的标识分别送到 T_S 和 $T_{S'}$ 的根结点；之后每个根结点广播更改层号和超结点标识消息到它们的每个结点，层号改成现有层号加一。若开始时 ID(S) > ID(S')，则从 u 到 T_S 的根结点路径上的父子关系颠倒过来，且 u 的父结点是 v；否则，将 v 到 $T_{S'}$ 的根结点路径上的父子关系颠倒一下，且 v 的父结点是 u。

5.6.3 算法的复杂性分析及正确性证明

定理5.9 算法5.8最终会结束。

证明 在算法 5.8 中，与每个结点 $v \in V$ 关联的边分为三类：无用边，树边、未检测边。算法初始化时，整个网络 $G(V,E)$ 的所有边都是未检测边，随着计算的进展，未检测边就渐渐地减少了，这些减去的边分别划归为树边和无用边类中。结果是，在每个超结点内，当一个树叶结点发现自己已没有未检测的边时，它立即向父结点发送“计算终止”的消息，当一个非叶结点（或称内部结点）收到所有儿子发送来的消息都是“计算终止”时，若它自己也找不出未检测边，它就向父结点（若存在的话）发送“计算终止”消息。最终的未检测边集合将成为空集，此时根结点将知道它的所有儿子及它自己都没有未检测边时，它向整个树结点发“终止执行”消息、算法终止。

定理 5.10 在基于异步通信的分布式计算模型上，找出一个加权的无向连通图 $G(V,E)$，$|V| = n$，$|E| = m$ 的最小生成树的分布式算法 5.8，其通信复杂性为 $O(m + n\log n)$，时间复杂性为 $O(n\log n)$。

证明 很显然，在任何计算 MST 的分布式算法中，$G(V,E)$ 中的每条边至少要搜索一次。因此任何 MST 分布式算法的通信复杂性下界是 $\Omega(m)$。在本节介绍的分布式算法中、为了确定每条边上最多传送的消息数目。我们观察到，算法结束时有 $m-(n-1)$ 条边是

无用边，在整个算法执行期间，每条无用边上只传递了常数条消息；另一方面，在每条树边上有可能传递了 $O(\log n)$ 条消息，因为最多有 $O(\log n)$ 个归并阶段，而每一归并阶段树边上传递的消息数目是一个常量。故整个算法的通信复杂性为 $O(m+n\log n)$。

同样，$\Omega(n)$ 是任何 MST 分布式算法的时间复杂性下界。由于在最坏情况下，在它的下一个处理机执行第一步之前，每个处理器至少要执行一步，也就是说，处理机初始化过程完全是顺序进行的。对这个算法的时间复杂性分析的关键是何时以及怎样对层号进行修改。本节算法是仅当具有相同层号超结点归并时才对层号进行修改，故整个算法至多有 $\lceil \log n \rceil$ 层，而每次归并过程树上操作时间至多为 $n-1$。故整个算法的时间复杂性为 $O(n\log n)$。

5.6.4 其它改进的分布式 MST 算法

Chin 等人[34] 曾对算法 5.8 进行了改进，在通信复杂性不变的前提下，给出了一个 $O(n\log^* n)$① 时间复杂性的算法，他们为了对超结点内的结点数目进行更准确的估计，使用了一个较层号更准确的指示器，这个指示器常常及时地反映超结点所含结点数目的多少。应该看到：算法 5.8 用层号估计超结点所含结点的数目，仅仅是一个下界，实际上一个超结点所含结点数目可以远远大于这个下界。因此，当 level_number(S') >level_number(S) 时，在 S' 与 S 归并之前，让 S' 等待也许是没有必要的。改进的算法则经常洞察超结点所含结点数目的实际大小，及时更新层号以反映超结点的真实情况，所以他们获得了一个时间复杂性为 $O(n\log^* n)$ 的改进算法。

最近 Awerbuch 进一步地把时间复杂性改进到 $O(n)$[35]，他的工作是使得层号更精确地反映超结点实际所含结点数目。在每一归并阶段，若检测到一个超结点的层号 l 小于这个结点所含的结点个数 n_s 的对数，则这个超结点的层号立即变为 $\lfloor \log n_s \rfloor$。

5.7 小　　结

本章着重介绍了求解图论上一个极其重要的基本问题 —— 最小生成树 (MST) 问题的许多并行算法。这些算法主要有：著名的 Sollin 算法在 SIMD – CREW PRAM 上、SIMD 二维网孔上以及在 MIMD 共享存贮模型上的并行化实现算法；Bentley 的树机上 MST 算法。同时还介绍了基于 SIMD – CREW PRAM 上的图更新维持 MST 性质的 MST 更新算法；最后介绍了分布式计算模型上的 MST 算法。

有关 MST 并行算法的研究，一直受到人们的高度重视，现已取得了不少成果。参见表 5.1。Chin 等人以及 Tsin 等人基于 Sollin 算法及 Hirschberg 顶点倒塌的思

注① $\log^* n = \min\{i \mid \underbrace{\log\log\cdots\log n}_{i\text{重对数}} \leq 1\}$

想，在 SIMD – CREW PRAM 上，分别建议了 $O(\log^2 n)$ 时间、$O(n^2/\log^2 n)$ 处理器的算法[3,17]。Kucera 基于 Kruskal 算法，在 SIMD – CRCW PRAM 上，建议了一个 $O(\log m)$ 时间、$O(mn^3)$ 处理器的算法[5]。Nath 等人基于 Sollin 算法。在 SIMD – EREW PRAM 上，给出了 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器的算法[20]。Kwan 等人则在 SIMD – CREW PRAM 上，考虑了稀疏图的 MST 算法，他们分别对 Sollin 算法、Prim – Dijkstra 算法以及 Kruskal 算法进行了并行化，最后得出算法复杂性分别为 $O(m\log n/p)$ 时间、$O(p)$ 处理器 $(p \leqslant m/\log n)$，$O(m\log n/p)$ 时间、$O(p)$ 处理器 $(p\log p \leqslant m\log n/n)$ 以及 $O(m\log n/p)$ 时间、$O(p)$ 处理器 $(p \leqslant \log n)$[22]，在互连网络模型上，Doshi 等人利用分而治之策略以及数据归约技术，在 SIMD 一维线性阵列上给出了 $O(n^2/p)$ 时间、$O(p)$ 处理器的算法 $(1 \leqslant p \leqslant n)$[24]。Yeh 在树机上给出了 $O(n^2/p)$ 时间、$O(p)$ 个处理器的算法[25]。Awerbuch 基于洗牌网络且每个处理器允许写冲突，给出了一个 $O(\log^2 n)$ 时间、$O(n^2)$ 处理器算法[26]。基于树网结构，Nath 等人曾建议了一个 $O(\log^3 n)$ 时间、$O(n^2)$ 处理器算法[21]，而后来 Huang 则给出了一个 $O(n^2/p)$ 时间、$O(p)$ 处理器的算法 $(p \leqslant n^2/\log^2 n)$[27]。Hambrush 在二维网孔上曾给出一个 $O(n)$ 时间、$O(n^2)$ 处理器算法[28]；最近 Maggs 等人基于二维网孔这种特殊结构，证明了找图的 MST 等价于找一个顶点为根的至各顶点的最短路径，给出了一个 $O(n)$ 时间、$O(n^2)$ 处理器的算法[29]。关于 MST 更新问题，最近 Jung 等人在 SIMD – CREW PRAM 上，获得了 $O(\log n)$ 时间、$O(n/\log n)$ 处理器的最优算法[30]。基于 MIMD 紧耦合异步共享存贮模型，Deo 等人曾对 Sollin 算法、Prim – Dijkstra 算法以及 Kruskal 算法进行了并行化，最后他们得出了 Sollin 算法最容易并行化的结论[31]。Yoo 也基于这种异步共享存贮模型，设计了一种适合于并行计算的数据结构 —— 并行堆结构，并引入了软件流水线概念，对 Kruskal 算法进行了并行化，给出了一个 $O(m)$ 时间、$O(\log m)$ 处理机算法[6]。最近作者基于 SIMD – CREW PRAM 及 SIMD – CRCW PRAM，给出了与 MST 及它的更新相关的另一个问题 —— 找 K 个 MST 的并行算法[32]。在这两种模型上，算法分别需 $O(\log^2 n + K\log n)$ 时间、$O(n^2)$ 处理器及 $O(K\log n)$ 时间、$O(n^2)$ 处理器。

有关基于异步通信的分布式计算模型上开发的 MST 算法已取得了一些成果。Dalal 从维护 MST 角度出发曾建议了另外一个 MST 分布式算法[36]；Humblet 曾考虑有向连通图的有向 MST 算法[37]；Spira 曾对 MST 分布式算法的通信复杂性进行了详细的分析[38]；另外 Parker 等人也从维护 MST 角度出发建议了另一个分布式算法[39]。

值得一提的是，Gallager 等人建议的 MST 分布式算法还具有方法学上的意义，Ramarao 曾利用这种设计分布式算法的思想设计了许多分布式算法[40]。

表 5.1 并行最小生成树算法

年 份	作 者	方 法	模 型	时间复杂性	处理器复杂性	备 注
1982	Chin 等人[3]	Sollin	SIMD–CREW PRAM	$O(\log^2 n)$	$O(n^2)$	
1982	Kucera[5]	Kruskal	SIMD–CRCW PRAM	$O(\log m)$	$O(mn^3)$	
1982	Nath 和 Maheshwari[20]	Sollin	SIMD–EREW PRAM	$O(\log^2 n)$	$O(n^2/\log n)$	
1983	Yoo[6]	Kruskal	MIMD 紧耦合异步共享	$O(m)$	$O(\log m)$	
1983	Nath 等人[21]	Sollin	SIMD–树网	$O(\log^3 n)$	$O(n^2)$	
1984	Tsin 和 Chin[17]	Sollin	SIMD–CREW PRAM	$O(\log^2 n)$	$O(n^2/\log^2 n)$	
1984	Kwan 和 Ruzzo[22]	Sollin Prim–Dijkstra Kruskal	SIMD–CREW PRAM	$O(m\log n/p)$	$O(p)$	$p\leqslant m/\log n$ $p\log p\leqslant m\log n/n$ $p\leqslant \log n$
1985	Huang[27]		SIMD 树网	$O(n^2/p)$	$O(p)$	$p\leqslant n^2/\log^2 n$
1987	Doshi 和 Varman[24]		SIMD 一维线性阵列	$O(n^2/p)$	$O(p)$	$1\leqslant p\leqslant n$
1987	Awerbuch 等人[26]		SIMD 洗牌网络	$O(\log^2 n)$	$O(n^2)$	
1987	Maggs 和 Poltkin[29]		SIMD 二维网孔	$O(n)$	$O(n^2)$	
1988	Jung 和 Mehlhorn[30]		SIMD–CREW PRAM	$O(\log n)$	$O(n/\log n)$	MST 更新
1990	唐策善和梁维发[32]		SIMD–CREW PRAM	$O(\log^2 n+k\log n)$	$O(n^2)$	求 k 个最小生成树
			SIMD–CRCW PRAM	$O(k\log n)$	$O(n^2)$	

说明：n 是图的顶点数；m 是图的边数；p 是处理器数目；k 是第 k 个 MST。

参 考 文 献

[1] Aho A V, Hopcroft J E, Ullman J D. *The Design and Analysis of Computer Algorithns*, Addison–Wesley, 1974

[2] Bentley J L, A Parallel Algorithm for Constructing Minimum Spanning Trees, *J Algorithms*, 1, 1980, 51–59

[3] Chin F Y , Lam J, Chen I N, Efficient Parallel Algorithms for Some Graph Problems, *Commu. ACM*, 25(9),1982, 659–665

[4] Savage C, Ja′ Ja′ J. Fast Efficient Parallel Algorithms for Some Graph Problems, *SIAM J. Comput.*, 10(4), 1981, 682–691

[5] Kucera L. Parallel Computation and Conflicts in Memory Access, *Inform. Proc. Lett.*, 14(2), 1982, 93–96

[6] Yoo Y B. Parallel Processing for Some Natwork Optimization Problems, Ph. D Thesis, Dept. of Computer Science, Washington State Univ., 1983

[7] Sollin M. *An Algorithm Attributed to Sollin*, in Introduction to the Design and Analysis of Algorithms, S.E.Goodman and S.T.Hedetniemi, McGraw–Hill, 1977

[8] Prim R C. Shortest Connection Networks and Some Generalizations, *Bell Syst. Tech. J.* 36, 1957, 1389–1401

[9] Dijkstra E. A Note on Two Problems in Connesion with Graphs, *Numer. Math.*, 1, 1959, 269–271.

[10] Kruskal J B. On the Shortest Subtree of a Graph and the Traveling Salesman Problem, *Proc. Amer. Math. Soci.*, 7, 1956, 48–50

[11] Savage C. Parallel Algorithms for Graph Theoretic Problems, Ph.D diss., Mathematics Dept., Univ. of Illinois at Urbana–Champaign, 1977

[12] Atallah M J, Kosaraju S R. Graph Problems on a Mesh– Connected Processor Array, *J. ACM*, 31(3), 1984, 649–667

[13] Atallah M J, Kosaraju S R. Graph Problems on a Mesh–Connedted Processor Array, *Proc. 14th Annu. ACM Sympo. on Theory of Computing, San Francisco*, Calif., NY., 1982, 345–353

[14] Nassimi D, Sahni S. Data Broadcasting in SIMD Computers, *IEEE Trans. Computer*, C–30(2),1981,101–106

[15] Pawagi S, Ramakrishnan I V. Parallel Update of Graph Properties in Logarithmic Time, *14th Intern. Conf. on Parallel Processing*, 1985

[16] Pawagi S, Ramakrishnan I V. An O(log*n*) Algorithm for Parallel Update of Minimum Spanning Trees, *Inform. Proc. Lett.*, 22, 1986, 223–229

[17] Tsin Y H, Chin F Y. Efficient Parallel Algorithms for a Class of Graph Theoretic Problems, *SIAM J. Comput.*, 13(3), 1984, 580–590.

[18] Vishkin U. Implementation of Simultaneous Memory Access in Models That Forbid it, *J. Algorithms*, 4, 1983, 45–50

[19] Quinn M J. *Designing Efficient Algorithms for Parallel Computers*, McGraw–Hill, 1987

[20] Nath D , Mahashwari S H. Parallel Algorithms for the Connected Components and Minimal Tree Problems, *Inform. Proc. Lett.*, 14(1), 1982, 7–11

[21] Nath D, Maheshwari S N, Bhatt P C P. Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees, *IEEE Trans. Computer*, C–32(6), 1983, 569–581

[22] Kwan S C, Ruzzo W L. Adaptive Parallel Algorithms for Finding Minimum Spanning Trees, *Proc. Intern. Conf. on Parallel Processing*, 1984, 439–443

[23] Quinn M J, Deo N. Parallel Graph Algorithms, *Computing Surveys*, 16(3), 1984, 319–348

[24] Doshi K A, Varman P J. Optimal Algorithm on a Fixed–Size Linear Array, *IEEE Trans. Computer*, C–36(4), 1987, 460–470

[25] Yeh D Y, Lee D T. Graph Algorithms on a Tree–Structured Parallel Computer, *BIT*, 24, 1984, 333–340

[26] Awerbuch B, Shiloach Y. New Conectivity and MSF Algorithms for Shuffle–Exchange Network and PRAM, *IEEE Trans. Computer.*, C–36, 1987, 1258–1263

[27] Huang M D. Solving Some Graph Problems with Optimal or Near–Optimal Speedup on Mesh–of –Trees Network, *Proc. 26th Annu. IEEE Sympo. On FOCS*, 1985, 232–240

[28] Hambrush S E. VLSI Algorithms for the Connected Component Problem, *In Proceedings of the 15th Conf. On Inform. Sci. and Syst.*, Baltimore, Md, 1981, 275–279

[29] Maggs B M, Plotkin S A. Minimum–Cost Spanning Trees as a Path–Finding Problem, *Inform. Proc. Lett.*, **26**, 1987 / 88, 291–293

[30] Jung H, Mehlhorn K. Parallel Algorithms for Computing Maximal Independent Sets in Trees and for Updating Minimum Spanning Trees, *Inform. Proc. Lett.*, **27**, 1988, 227–236

[31] Deo N, Yoo Y B. Parallel Algorithms for the Minimum Spanning Tree Problem, 1981 *Intern. Conf. on Parallel Processing*, 1981, 188–189

[32] 唐策善，梁维发．找k个最小生成树的并行算法，中国科学技术大学学报，**20** (4)，1990，464–471

[33] Gallager R G, Humblet P A, Spring P M. A Distributed Algorithm for Minimum–Weight Spanning Trees, *ACM TOPLS*, 5(1), 1983, 66–77

[34] Chin F, Ting H F. An Almost Linear Time and O(*n*log*n*+*e*) Messages Distributed Algorithm for Minimum–Weight Spanning Trees, *Proc. 26th IEEE FOCS*, 1985, 257–266

[35] Awerbuth B. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and Related Problem, *Proc ACM STOC*, 1987, 230–240

[36] Dalal Y K. Broadcast Protocols in Packet Switched Computer Networks, Ph. D diss. Digital Syst. Lab., Stanford Univ., CA, TR–128, 1977

[37] Humblet P A. A Distributed Algorithm for Minimum Weight Directed Spanning Trees, *IEEE Trans. Commu.*, Com–31(6), 1983, 756–762

[38] Spira P M. Communication Complexity of Distributed Minimum Spanning Trees Algorithms , *in Proc. 2nd Berkeley Conf. Distributed Data Managment Comput. Networks*, 1977

[39] Parker D S, Samadi B. Adaptive Distributed Minimal Spanning Tree Algorithm, *in Proc. 1st Sympo. Reliability Distrib. Soft. Database*, 1981

[40] Ramarao K V S. Distributed Algorithms for Network Recognition Problems, *IEEE Trans. Compt.*, C–38(9), 1989, 1240–1248

第六章　最短路径的并行算法

在运输网络或通信网络中，最短路径问题有着极其重要的应用背景。本章介绍两类最短路径问题的并行算法：(1) 在一个网络图中，寻找从一个指定顶点到其它所有顶点间的最短路径——**单源最短路径问题**的并行算法；(2) 在一个网络图中，寻找每一对顶点间的最短路径——**所有顶点对的最短路径问题**的并行算法。在一个有向网络图 G (V, E) 中，只要不存在负权环，图中边的权值可以是正数也可以是负数。若存在负权环，则将使得对某些顶点间的最短路径的定义没有意义。在一个无向网络图中，边是无方向的，遍历时可以沿着边所连接的两个顶点中的任一个穿过该条边，但边的权值不能为负。

6.1　单源最短路径算法

单源最短路径问题是指求从一个指定的顶点 S 到其它所有顶点 i 之间的距离 dist(i) 为最短的路径，其中 $s \in V$ 称为源点，$i \in V$ 且 $i \neq s$。假定图 $G(V,E)$ 是一个有向网络，其加权邻接短阵为 W，且边上权值 $w(i,j)>0$，$i, j \in V$，$V=\{1, 2, \cdots, n\}$。

著名的 Dijkstra 算法[2] 的基本思想是：假定有一个待搜索顶点表 VL，初始化时做：dist(s)←0；dist(i)←∞ ($i \neq s$)；$VL \leftarrow V$。待加顶点表，每次从 $VL(\neq \phi)$ 中选取这样一个顶点 u，它的 dist(u) 值最小。将选出的顶点为搜索顶点，若 $<u,v> \in E$，而且 dist(u) + $w(u,v)$ < dist(v)，则更新 dist(v) 为 dist(u) + $w(u,v)$，直到 $VL = \phi$ 时算法终止。

算法 6.1　DIJKSTRA ALGORITHM (SISD)

输入：加权邻接短阵W，约定 i, j之间无边连接时$w(i,j)=\infty$，且$w(i,i)=\infty$；

输出：dist(1:n)，其中dist(i)表示顶点 s 到顶点 i 的最短路径($1 \leqslant i \leqslant n$)。

```
begin
    /*初始化*/
(1) dist(s)←0;
(2) for i←1 to n do
        if i≠s then dist(i)←∞ endif
    endfor;
(3) VL←V;
(4) for i←1 to n do      /*找最短距离*/
(5)     find a vertex u∈VL, such that dist(u) is minimal;
(6)     for each (<u,v>∈E) ∧ (v∈VL) do
            if dist(u)+w(u,v)<dist(v) then dist(v)←dist(u)+w(u,v) endif
```

```
        endfor ,
(7)     VL ← VL − {u}
      endfor
end
```

Paige 等人[1] 基于 SIMD − EREW PRAM，对算法 6.1 进行了并行化．为了简化讨论起见，约定顶点 i 的出度为 $d_{out}(i)$ $(1 \leqslant i \leqslant n)$．具体算法并行化如下：

算法的第 (2) 步的实现为：每个处理器分派 $\lceil n/p \rceil$ 个顶点，最后一个处理器分派 $n - \lceil n/p \rceil \cdot (p-1)$ 个顶点，每个处理器对所分派的顶点顺序地赋值；第 (3) 步其实是对数组 VL 赋值，同第 (2) 步类似处理；第 (5) 步实现如下：首先每个处理器检测自己内部 $\lceil n/p \rceil$ 个顶点的最小值，然后 p 个处理器合作，并行计算 p 个最小值中的最小值；第 (6) 步实现如下：首先将顶点 u 广播到所有的处理器中，假定 u 是送入处理器 i 的 $B(i)$ 单元中 $(1 \leqslant i \leqslant p)$，则实现广播的算法为：

算法 6.2 BROADCAST

```
输入：数据 u（存放在单元B(1)中）；
输出：将 u 广播到数组 B 的所有单元中去。
begin
(1) B(1) ← u ;
(2) for i ← 0 to ⌈logp⌉ − 1 do
(3)   for each j: 2^i + 1 ≤ j ≤ 2^(i+1) pardo
          B(j) ← B(j − 2^i)
      endfor
    endfor
end
```

然后每个处理器检查所有边 $<u,v> \in E$，其中 $v \in VL$，且 v 是本处理器的顶点．

定理 6.1 在 SIMD − EREW PRAM 上，求一个有向连通加权图 $G(V,E)$，$|V| = n$ 的单源最有边明 的算法 6.1 的并行化实现，需 $O(n^2/p + n\log p)$ 时间、$O(p)$ 处理器 $(1 \leqslant p \leqslant n)$．

证明 Dijkstra 算法并行化的每一步的复杂性分析如下：第 (1) 步需 $O(1)$ 时间在单个处理处理器边　源 (2) 和第 (3) 步需 $O(n/p)$ 时间、$O(p)$ 个处理器；第 (5) 步需 $O(n/p + \log p)$ 时间、$O(p)$ 处理器；第 (6) 步广播需 $O(\log p)$ 时间、$O(p)$ 处理器；然后进行比较更新，需 $O(d_{out}(u)/p)$ 时间、$O(p)$ 处理器，故第 (4) 步需

$$O(n(n/p + 2\log p) + \sum_{u \in V} \lceil d_{out}(u)/p \rceil) = O(n^2/p + n\log p + m/p + n)$$

时间、$O(p)$ 处理器．因此，整个算法需 $O(n^2/p + n\log p)$ 时间、$O(p)$ 处理器 $(1 \leqslant p$

$\leqslant n$).

Paige 等人还把上述在 SIMD – EREW PRAM 上的算法推广到 SIMD – CRCW PRAM 上，在这种计算模型上用 p 个处理器对 n 个元素进行二元结合运算（如求最小值运算）仅需 $O(n/p+\log\log p)$ 时间[3]，为此不难推得:

推论 6.1 在 SIMD – CRCW PRAM 上，求一有向加权图 $G(V,E)$，$|V|=n$ 的单源最短路径需 $O(n\log\log n)$ 时间和 $O(n/\log\log n)$ 处理器.

证明 类似定理 6.1 的证明，不再赘述.

6.2 所有顶点对的最短路径算法

设一个有向图 $G(V,E)$，$|V|=n$ 的加权邻接矩阵为 W，且 $w(i,j)$ 表示边 (i,j) 的长度，若 i 和 j 之间不存在有向边，则 $w(i,j)$ 为 ∞，$1\leqslant i,j\leqslant n$，$i,j\in V$．两个矩阵 A 和 B 的积 $C=AB$ 定义为:

$$C(i,j)=\min\{w(i,k)+w(k,j)\mid k=1,2,\cdots,n\}$$

令最短路径矩阵为D，则 Floyd 算法如下:

算法 6.3 FLOYD ALGORITHM (SISD)

输入: 有向图$G(V,E)$的加权邻接矩阵$W=\{w_{ij}\}$，$i,j\in V$；

输出: 所有顶点对的最短路径矩阵$D=\{d_{ij}\}$，$i,j\in V$.

```
begin
(1) D ← W ;
(2) for k ← 1 to n do
(3)     for i ← 1 to n do
(4)         for j ← 1 to n do
(5)             d(i,j ) ← min{d(i,j ), d(i,k) + d(k,j )}
            endfor
        endfor
    endfor
end .
```

对算法 6.3 人们很容易并行化，当使用 $p\leqslant n^2$ 个处理器时，我们可以对算法中的第 (1) 步和第 (3) 到 (5) 步并行化.

引理 6.1 在 SIMD – CREW PRAM 上，求一个有向加权图 $G(V,E)$，$|V|=n$ 的所有顶点对的最短路径，算法 6.3 的并行化实现，需 $O(n^3/p)$ 时间、$O(p)$ 处理器 ($1\leqslant p\leqslant n^2$).

证明 对算法 6.3 的并行化，数组 D 有 n^2 个元素，故给 n^2 个元素赋值需 $O(n^2/p)$ 时间、$O(p)$ 处理器；第 (3) 步到 (5) 步有 n^2 个 $d(i,j)$ 要计算，对这 n^2 个数进行检查

需 $O(n^2/p)$ 时间、$O(p)$ 处理器；而且第 (3) 到 (5) 步执行了 n 次，故整个算法需 $O(n^3/p)$ 时间、$O(p)$ 处理器 $(1 \leqslant p \leqslant n^2)$。

事实上，算法 6.3 也就是矩阵自乘 n 次。为此，求所有顶点对的最短路径并行算法亦就变为矩阵相乘的并行算法。

令 $G(V,E)$ 的最短路径矩阵为 D，同时令 $d^{(k)}(i,j)$，表示顶点 i 到顶点 j 之间的一条最短路径，且 这 条 路 径 至 多 经 过 了 2^k 个 中 间 顶 点，因 G 中不存在负环，所以 $d(i,j) = d^{\lceil \log n \rceil}(i,j)$。显然 $d^{(0)}(i,j) = w(i,j)$。在并行环境里，对所有 i, j 和同一个 k 值，$d^{(k)}(i,j)$ 同时进行计算 $(1 \leqslant i,j \leqslant n,\ 0 \leqslant k \leqslant \lceil \log n \rceil)$。

算法6.4 ALL VERTICES SHORTEST PATH ALGORITHM

输入：有向图 $G(V,E)$ 的加权邻接矩阵 $W = \{w_{ij}\}$，$i, j \in V$；

输出：所有顶点对的最短路径矩阵 $D = \{d_{ij}\}$，$i,j \in V$。

```
begin
/ * 并行赋初始值 * /
(1) for each i, j: 1 ≤ i,j ≤ n pardo
        d^(0)(i, j) ← w(i, j)
    endfor ;
/ * 并行计算所有顶点对的最短路径 * /
(2) for k ← 1 to ⌈logn⌉ do
(3)     for each i, j, l : 1 ≤ i, j, l ≤ n pardo
(4)         B(i, j, l ) ← d^(k-1)(i, l ) + d^(k-1)(l, j) ;
(5)         d^(k)(i, j) ← min{d^(k-1)(i, j), B(i, j, l ) | 1 ≤ l ≤ n}
        endfor
    endfor
end .
```

在给出上述算法的复杂性分析之前，首先我们证明上述算法的正确性。

引理 6.2 在加权邻接矩阵 W 赋给 $D^{(0)}$ 之后，每次结果矩阵需自乘 $\lceil \log n \rceil$ 次后才能求得 $D^{(n)}$。

证明 这里采用归纳法证明。设 $d^{(k)}(i,j)$ 表示顶点 i 到顶点 j 之间至多经过 2^k 个中间顶点的最短路径。当 $k = 0$ 时，$D^{(0)} = W$，命题显然成立。

归纳假定：在 $l < k$ 时，令 $d^{(l)}(i,j)$ 表示顶点 i 到顶点 j 之间至多经过 2^l 个中间顶点的最短路径。在算法 6.4 的第 k 次循环时，

$$d^{(k)}(i, j) \leftarrow \min\{d^{(k-1)}(i, j),\ B(i, j, l) \mid 1 \leqslant l \leqslant n\}$$

若 $d^{(k)}(i, j) = d^{(k-1)}(i, j)$，则表示顶点 i 到顶点 j 之间的最短路径不超过 $2^{(k-1)}$ 个；若存

在某个 $l_0(1 \leqslant l_0 \leqslant n)$，使得

$$d^{(k)}(i,j)=B(i,j,l_0)=d^{(k-1)}(i,l_0)+d^{(k-1)}(l_0,j)$$

由归纳假定可知：顶点 i 到顶点 l_0 之间至多有 $2^{(k-1)}$ 个中间顶点，顶点 l_0 到顶点 j 之间至多有 $2^{(k-1)}$ 个中间顶点，故顶点 i 到顶点 j 之间至多有 2^k 个中间顶点。又因为图 G 的最长路径至多有 $n-1$ 个中间顶点，所以 $k \leqslant \lceil \log n \rceil$。即由矩阵 $D^{(0)}$ 自乘 $\lceil \log n \rceil$ 次能够算出 $D^{(n)}$。

定理 6.2 算法 6.4 能正确地计算出所有顶点对的最短路径。

证明 由引理 6.2 直接推证。

定理 6.3 在 SIMD-CREW PRAM 上，求一个有向、无负环的加权图 $G(V,E)$，$|V|=n$ 的所有顶点对的最短路径算法 6.4，需 $O(\log^2 n)$时间、$O(n^3)$处理器。

证明 从算法 6.4 可知，第(1)步需 $O(1)$时间、$O(n^2)$处理器；第(3)到(5)步需 $O(n^3)$处理器，且第(4)步需 O(1)时间，第(5)步需 $O(\log n)$时间；由第(2)步可知，第(3)到(5)步需执行 $\lceil \log n \rceil$次。因此算法 6.4 需 $O(\log^2 n)$时间、$O(n^3)$处理器。

推论 6.2 在 SIMD-CREW PRAM 上，求一个有向、无负环的加权图 $G(V,E)$，$|V|=n$ 的所有顶点对的最短路径需 $O(\log^2 n)$时间、$O(n^3/\log n)$处理器[5]。

证明 在算法 6.4 的第(3)到(5)步采用分组技术，每个处理器每次顺序处理至多 $\lceil \log n \rceil$个元素。其余证明类似定理 6.3。

推论 6.3 在 SIMD-CRCW PRAM 上，求一有向的、无负环的加权图 $G(V,E)$，$|V|=n$ 的所有顶点对的最短路径只需 $O(\log n)$时间、$O(n^4)$处理器。

证明 采用 4.1 节的 Kucera 方法[6]，矩阵乘积运算可在 $O(1)$时间内完成，而由引理 6.2，每次结果矩阵必须自乘 $\lceil \log n \rceil$次，故算法的复杂性不难推得。

6.3 二维网孔上的最短路径算法

Guibas 等人[7] 曾在 $n \times n$ 个处理器的二维 **Systolic**（心动）网孔上给出了一个计算图 $G(V,E)$ 的传递闭包算法，这一算法的重要应用背景就是求有向加权图的所有顶点对的最短路径。令 $G(V,E)$ 的邻接矩阵为 A。

算法6.5 TRANSITIVE CLOSURE ON SYSTOLIC ARRAY

输入：有向或无向图的加权邻接矩阵 $A=\{a_{ij}\}$，$i,j\in V$；

输出：矩阵 A 的传递闭包矩阵 $C=\{c_{ij}\}$，$c(i,j)$ 存放在编号为 (i,j) 的处理器寄存器中，$i\in V$，$j\in V$。

begin

(1) 给矩阵 A 的传递闭包赋初值。$c(i,j)\leftarrow a(i,j)$，同时 $a(i,j)$ 有两个副本，一个作水

平移动，称作$a^{(1)}(i,j)$，另一个作垂直移动，称作$a^{(2)}(i,j)$；

(2) 在某一时钟节拍，若元素$a^{(1)}(i,k)$及$a^{(2)}(k,j)$同时进入编号为(i',j')的处理器，此时这个处理器做：

(i) 若$i=i'$且$k=j'$，则$a^{(1)}(i,k)\leftarrow c(i',j')$；

(ii) 若$k=i'$且$j=j'$，则$a^{(2)}(k,j)\leftarrow c(i',j')$；

(iii) $c(i',j')\leftarrow c(i',j')\vee(a^{(1)}(i,k)\wedge a^{(2)}(k,j))$；

(3) 数据按图6.1所示方向流动，直到$a^{(1)}(n,n)$和$a^{(2)}(n,n)$穿过整个网孔为止

end.

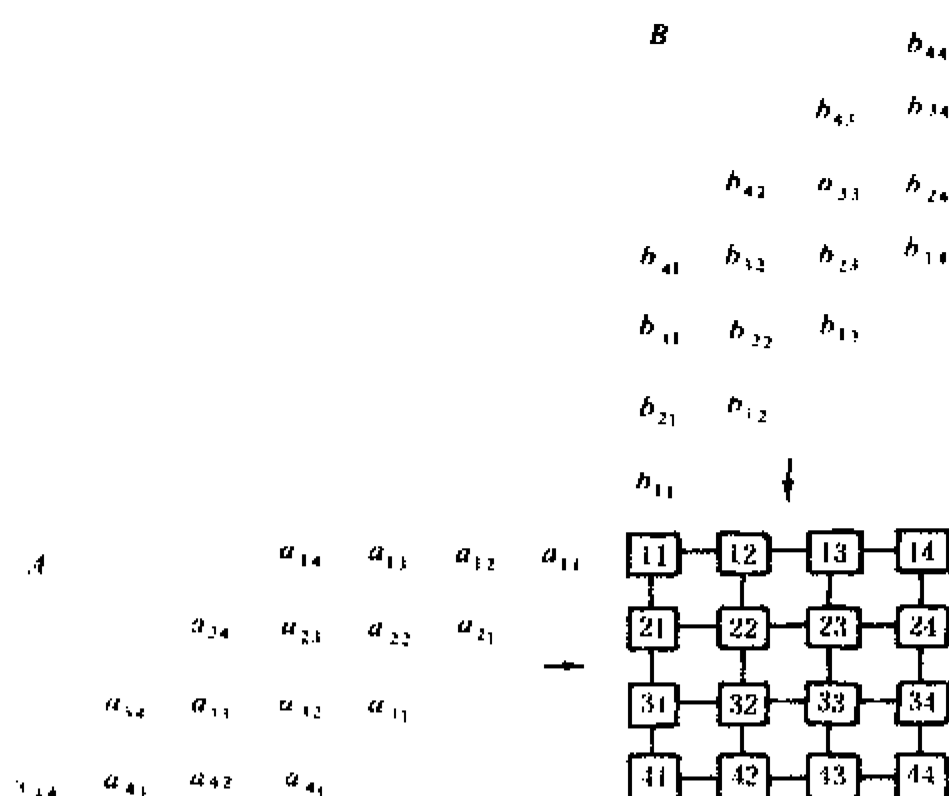


图 6.1 网孔上的矩阵乘法

由此可知：从$a^{(1)}(1,1)$进入网孔到$a^{(1)}(n,n)$离开网孔需$O(n)$时间。

定理 6.4 在SIMD二维心动网孔上，计算布尔矩阵A的传递闭包，算法6.5需$O(n)$时间、$O(n^2)$处理器。

证明 显而易见。

推论 6.4 在SIMD二维心动网孔上，求一个有向无负环的加权图中所有顶点对的最短路径，需$O(n)$时间、$O(n^2)$处理器。

证明 把算法6.5中的A换成加权邻接矩阵W，C为所有顶点对的最短路径矩阵，算法6.5中第(2)步的(iii)换成：

$$c(i',j')\leftarrow\min\{c(i',j'),\quad w^{(1)}(i,k)+w^{(2)}(k,j)\}$$

则得到的算法就是一个求所有顶点对的最短路径的算法．且这样的转换并不影响算法的复杂性，故推论得证．

6.4 MIMD共享存贮模型上的最短路径算法

Deo等人[8]基于MIMD紧耦合共享存贮模型，实现了Moore算法的并行化．为了介绍他们的算法，我们先介绍Moore单源最短路径算法[9]．

在Moore单源最短路径算法中，设源点为$s\in V$，从s到其它各顶点的最短路径长度用一个一维数组dist存贮．首先置$\text{dist}(s)=0$，$\text{dist}(v)\leftarrow\infty$，$v\neq s$，$v\in V$．Moore算法同Dijkstra算法不同之处是：将Dijkstra算法中的待搜索顶点表替换成待搜索队列．算法

开始执行时，队列仅含源点 s。以后每次只要待搜索队列非空，则将队头的顶点从队列中移出作为本次搜索的顶点，然后检查 u 的所有射出边 $<u,v>\in E$。若 $\text{dist}(u)+w(u,v)<\text{dist}(v)$，则此时找到了一条从 s 到 v 的更短路径，置 $\text{dist}(v)$ 等于 $\text{dist}(u)+w(u,v)$。若 v 不在待搜索队列中，则把 v 加入到队列的队尾。如此重复进行，直到整个待搜索队列空时，算法终止。下面我们给出单机上的 Moore 算法。

算法6.6 MOORE ALGORITHM (SISD)

输入：有向图 $G(V,E)$ 的加权邻接矩阵 $W=\{w_{ij}\}$，$i,j\in V$；

输出：从源 s 到所有其它顶点 i $(i\neq s)$ 的最短路径 $\text{dist}(i)$，$i\in V$。

```
begin
(1) for i←1 to n do        /* 初始化 */
        call INITIALIZE(i)
    endfor;
(2) insert s into the queue;     /* 插入 s 到队列中 */
(3) while the queue is not empty do
(4)     call SEARCH
    endwhile
end.
procedure SEARCH;
  begin
  (4.1) dequeue vertex u;         /* 从队列中删去顶点u */
  (4.2) for each edge (u,v)∈E do
  (4.3)     new_dist←dist(u)+w(u,v);
  (4.4)     if new_dist<dist(v) then dist(v)←new_dist  endif;
  (4.5)     if v is not in the queue  then enqueue vertex v  endif
        endfor
  end.
```

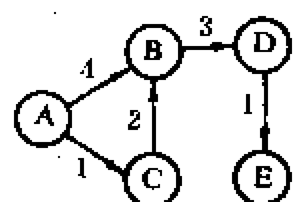
其中，过程 INITIALIZE 是做初始化工作，即是做：$\text{dist}(s)\leftarrow 0$；$\text{dist}(v)\leftarrow\infty$ $(v\neq s, v\in V)$；$\text{queue}\leftarrow\phi$。

现在我们用一个具体例子来说明算法 6.6 的执行过程，如图 6.2 所示。

6.4.1 算法的基本原理

Deo 等人的算法是算法 6.6 的并行化。直观上讲，算法 6.6 有两种明显的并行化方法。第一种方法是对 SEARCH 过程的第(2)到(5)步并行化。任何一个顶点，可能存在有几条射出边，这些射出边可以并行地检查。第二种方法是对算法的第(3)到(4)步的 **while** 循环并行化。在算法执行过程中，任何时候都可能有许多顶点在队列中，因此，每次就可能检查多个顶点的射出边。上述两种并行化方法，究竟哪一种更好呢？我们完全有理由赞成

使用第二种方法。首先，第二种方法能产生**大粒度**(Larger Grained)任务进程。根据粒度尺寸定理[16]，它很可能产生好的加速比；其次，第一种方法的并行度受到搜索顶点的射出边多寡的约束。若图 G 是一个稀疏图时，则其活动的进程数就少。

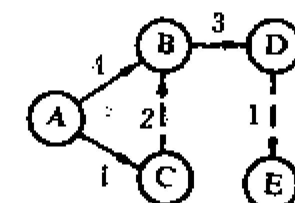


图G(V、E)

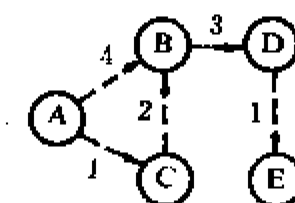
6.4.2 算法的形式化描述

我们用产生大粒度的、适合于稀疏图的第二种方法来描述并行算法。首先队列用源点初始化。然后创建了许多异步进程。每个进程都执行下列步骤：即从队列中删除一个顶点，检查被删除顶点的射出边，将已发现的、有更短路径的顶点加入到队列中，算法 6.6 的第(1)步采用**伪调度**(Prescheduling)方法很容易并行化。第(3)步和(4)步的 **while** 循环必须做适当的修改，使得能反映并行执行 SEARCH 过程时许多异步进程的存在性。显然，当一个进程发现队列为空时，这个进程就终止执行是不适当的。因而必须采用更复杂的办法。在下面描述的算法中，我们将两个变量联合使用，以便决定还有没有工作要做。第一个变量是数组变量 waiting，它记住哪一个进程正处在等待状态(等待工作去做)；第二个变量是布尔变量 halt，仅当队列为空和所有进程处于等待状态时为真，且 INITIALIZE 过程置数组 waiting 中的每一个元素为假。

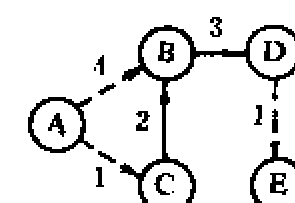
SEARCH 过程也必须作适当的修改。因为对队列的插入，删除操作不是原子(Atomic)操作，所以当一个元素加到队列中去或从队列中删除一个元素时，必须给队列上锁。其次，在一个进程将刚发现到 v 路径 new_dist 与当前的最短路径 dist(v)比较之前，变量 dist(v)必须上锁，否则，两个进程有可能同时修改 dist(v)，得到的结果是不正确的。如图 6.3 所示。最后，若一个进程发现队列为空时，则置数组 waiting 中的相应元素为真。若进程 1 处在等待状态，则它检查每个进程是否都在等待状态。如果每个进程都在等待，那么 halt 值置



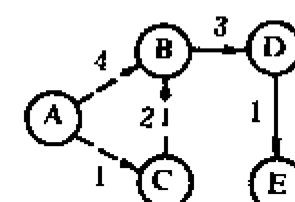
距离		队列
A	0	A
B	∞	
C	∞	
D	∞	
E	∞	



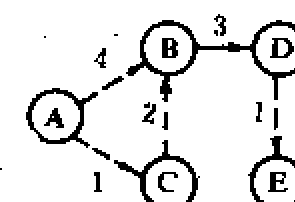
距离		队列
A	0	B
B	4	C
C	1	
D	∞	
E	∞	



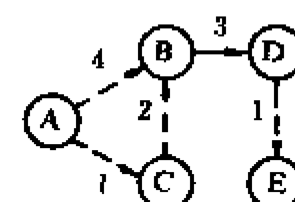
距离		队列
A	0	C
B	4	D
C	1	
D	7	
E	∞	



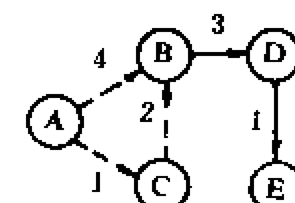
距离		队列
A	0	D
B	3	B
C	1	
D	7	
E	∞	



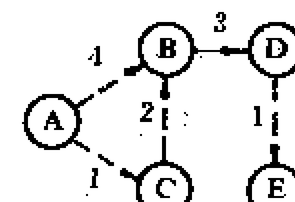
距离		队列
A	0	B
B	3	E
C	1	
D	7	
E	8	



距离		队列
A	0	E
B	3	D
C	1	
D	6	
E	8	



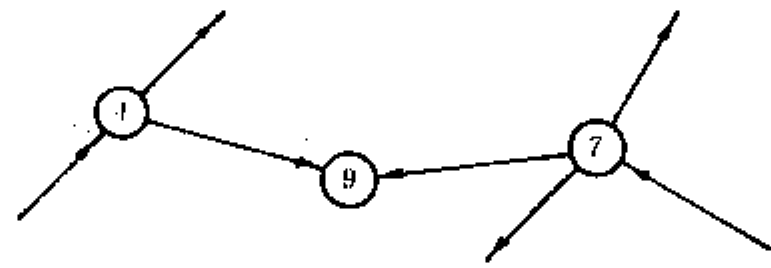
距离		队列
A	0	D
B	3	
C	1	
D	6	
E	8	



距离		队列
A	0	E
B	3	
C	1	
D	6	
E	7	

图 6.2 Moore 的单源最短路径算法示例

成真．注意：在进程 1 检查每个进程是否都处于等待状态时，队列必须上锁．



dist (9) 的当前值	进程 1	进程 2
28	从队列中删去顶点 4	从队列中删去顶点 7
28	考虑边(4,9)	考虑边(7,9)
28	new_dist←22	new_dist←24
28	new_dist < dist (9)	new_dist < dist (9)
28	dist (9)←22	•
22		dist (9)←24
24		•

图 6.3 不用锁时，两个进程可能企图同时修改 dist(v)值引出的错误

算法6.7 PARALLEL ALGORITHM FOR SHORTEST PATH

输入：有向图$G(V,E)$的加权邻接矩阵$W=\{w_{ij}\}$，i，$j\in V$；

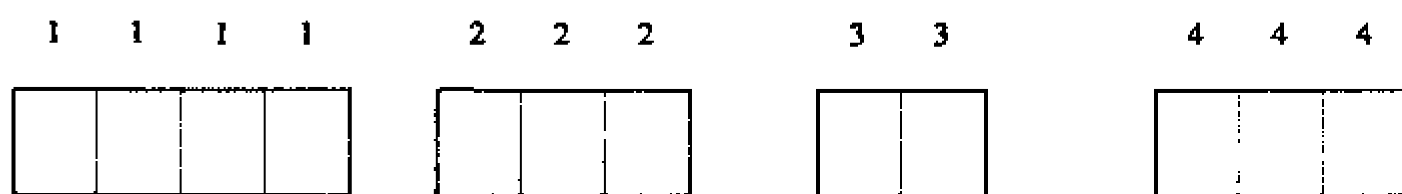
输出：从源 s 到其它所有顶点 i 的最短路径 dist(i)，$i\neq s$，$i\in V$．

```
begin
(1) for each i: 1≤i≤p pardo
        for j←i to n step p do   /*初始化*/
            call INITIALIZE(j)
        endfor
      endfor ;
(2) enqueue s ;      /*从队列中删去s*/
(3) halt←false ;     /*置布尔变量为假*/
(4) for each i: 1≤i≤p pardo
(5)       while not halt do
(6)             call SEARCH(i)
          endwhile
      endfor
end.
procedure SEARCH(i);
begin
(6.1) lock the queue;      /*队列上锁*/
```

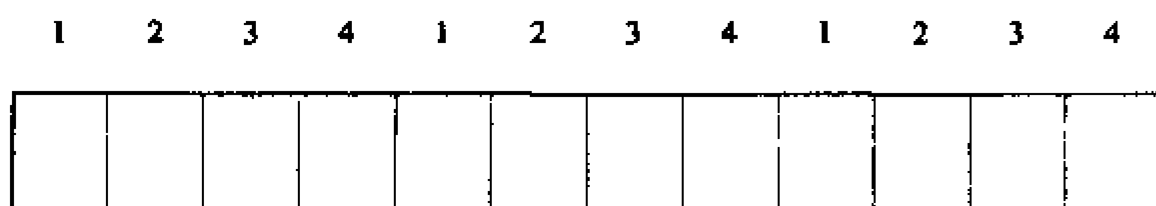
```
(6.2) if the queue is empty      /* 队列空时置相应的元素为真 */
(6.3)    then waiting(i) ← true;
(6.4)         if i = 1      /* 进程1等待时，检查其它进程是否在等待 */
                 then halt ← waiting(2) and waiting(3) and··· and waiting(p)
              endif;
(6.5)         unlock the queue      /* 队列开锁 */
(6.6)     else dequeue;  /* 从队列中删除u */
(6.7)         waiting(i) ← false;
(6.8)         unlock the queue;
(6.9)         for each edge <u,v> ∈ E do      /* 检查每条射出边 */
(6.10)            new_dist ← dist(u) + w(u,v);
(6.11)            lock(dist(v));
(6.12)            if new_dist < dist(v)
(6.13)               then dist(v) ← new_dist;    /* 更新new_dist */
(6.14)                    unlock(dist(v));
(6.15)                    if v∉queue
(6.16)                       then lock the queue;
(6.17)                            enqueue v;
(6.18)                            unlock the queue
                          endif
(6.19)               else unlock (dist(v))
                 endif
             endfor
      endif
  end.
```

值得指出的是：虽然创建更多的进程能降低整个算法的执行时间，因为几个顶点的射出边可以同时进行检查。但每个进程对队列的插入及删除需进行互斥控制，故最大加速比最终还是受到限制的.

若每个进程维护一个局部的待搜索表，插入和删除元素都在局部的表中进行，那么进程间就不存在冲突了. 然而，若这种表的尺寸变化很大时，则让每个进程管理自己的表格会引起工作严重的负载不平衡. 一种折衷的方法是：每个进程插入元素到自己的局部表中，然后将这些局部表连接起来形成一个结合表. 删除可以这样做：p个进程的每一个 i 仅检查结合表的第$(i-1)\cdot p+1$ 个元素. 按这种方法去作，每个进程所做的工作就平衡了. 如图 6.4 所示.



(a) 每个处理器在自己的空间内插入——无冲突



(b) 每个处理器隔 p 个元素删除——无冲突

图 6.4 链接数组的逻辑形式

链接数组 (Linked Array) 是一种数据结构[15]. 设计它的目的是可以将不同尺寸的表链接起来，且对表元素的插入和搜索可以并行地进行而不发生冲突. 假定在单个循环步中，每个进程至多插入 w 个元素. 在这种假定下，链接数组含有 $p\cdot(w+p)$ 个元素，而每个进程有 $w+p$ 个元素，间隔为 $w+p$ 位置后面的连续一组元素是进程下一次循环时待搜索的元素. 如果进程 i, $1\leqslant i\leqslant p$ 产生了 e_i 个元素，且将在下一次循环时才会被搜索，那么，从 $(i-1)\cdot(w+p+1)+1$ 到 $(i-1)\cdot(w+p+1)+e_i$ 处含有这 e_i 个元素，从 $(i-1)\cdot(w+p+1)+e_i+1$ 到 $(i-1)\cdot(w+p+1)+p$ 位置上分别含有值 $-i\cdot(w+p+1)$ 到值 $-i\cdot(w+p+p)$.

在下一次循环时，当位于链接数组的元素需要检索时，则处理器 i, $1\leqslant i\leqslant p$ 从位置 i 开始，每隔 p 个位置执行一次检查. 若检索到的值大于 0, 则就是待搜索的顶点元素；若检索到的值小于 0, 则这个值是一个指针，处理器立即检查指针所指位置，当指针值小于 $-p\cdot(w+p)$ 时，搜索终止.

过程 EXAMINE 说明一个进程如何从一个含有顶点元素和指针的数组中移去一个元素.

算法 6.8 EXAMIZATION TO ELEMENTS

```
procedure EXAMINE (A, i, p, w);
/ * A是含顶点元素和指针的一维数组, p是进程数, 每个进程含w + p个元素, 移
    去 i 中元素 * /
begin
(1)  j← i;
(2)  while j ≤ p(w + p) do
(3)     if a(j) < 0 then j← - a(j)
                      else manipulate vertex whose value is a(j);
                           j← j+ p
        endif
```

```
        endwhile
    end .
```

从空间开销来讲，使用链接数组可能是昂贵的，除非保证不是单个处理器进行插入。链接数组需分配的空间大约是每个处理器子表空间的 p 倍。算法 6.6 的一个并行划分版本可设计成使用链接数组结构。在一次循环中，每个进程有许多顶点将被检查，每个进程将编辑它自己的顶点表，这些顶点是到目前为止已发现的更短路径顶点，然后每个进程建立到下一个进程的链。在下一次循环时，并行地检查这些表。算法使用了两个数组，在任何一次循环中，一个数组用于读，而另一个数组用于写，紧接着的下一次循环，两个数组的作用再颠倒过来。

6.5 分布式单源最短路径算法

Chandy 等人[17]曾考虑了带负环的单源最短路径算法。本节我们介绍这一算法。已知一个有向加权图$G(V,E)$，$V=\{1, 2, \cdots, n\}$，边 $<i, j>\in E$ 的权值 $w(i, j)$可以为正，也可以为负。若 i 与 j之间不存在边，则$w(i, j)=\infty$。约定 $w(i, i)=0$, $(1\leqslant i, j\leqslant n)$。设 s 为源点，s 到 i 的最短路径为L_i。若图 G 中存在一个负环 C，则 C 上所有结点 i 的 L_i 为$-\infty$。假定每个结点$i\in V$知道其所有射出边的邻接结点和这些射出边上的权值。

6.5.1 算法的基本原理

算法基本上分为两个阶段。每一个阶段开始时，都是源点 s 作本阶段的初始化工作。在第一阶段结束时，对每个结点 i 而言，若 $L_i\neq-\infty$，则它得到一个有限的最短路径 L_i。若存在某个结点 j $(j\neq s)$ 有 $L_j=-\infty$，则在第一阶段结束时 j 自己并没有注意到这一点。第二阶段的目标就是通知所有 $L_i=-\infty$ 的结点 i 停止执行，且告诉它们是负环可达的结点。

算法的第一阶段使用了两种消息，即有关最短路径长度消息和应答消息。用二元组 (l, i) 表示最短路径长度，其中：l 是一个实数，i 是结点的编号。结点 i 发送一个 (l, i) 消息给结点 j，它通知 j 从源 s 到 j 存在一条长度为 l 的最短路径，i 是这条路径上的最后一个中间结点。

算法的第二阶段使用了 "over ?" 及 "over −" 两种消息。若一个结点 j 确信它的第一阶段已经完成，且 $L_j=-\infty$，则 j 向 $N_{out}(j)$ 所有成员发送 "over −" 消息。"over −" 消息的接收者一旦收到该消息后，就停止第一阶段的计算，并将 "over −" 消息拷贝到它的射出边的每一个邻接结点。当某个结点 j 第一阶段已经完成，但还没有确定 $L_j=-\infty$ 时，则它向 $N_{out}(j)$ 发送 "over ?" 消息。"over ?" 消息命令接收者停止第一阶段的计算。若 "over ?" 消息接收者在收到此消息时，它已收到所有应答消息，则它拷贝 "over ?" 到每个射出边的邻接结点；否则，发送 "over −" 至每个射出边的邻接结点。

6.5.2 算法的形式化描述

算法用到下面的一些局部变量:

d_i: 到目前为止，结点 i 收到的最短路径长度。初值 $d_i=\infty$ $(i\in V)$;

father(i): 接收到 d_i 的发送者。当 $d_i=\infty$ 或 $i=s$ 时，father(i) $=i$ $(i\in V)$;

num_i: 结点 i 发出的消息数目，即到目前为止尚未收到应答的消息数目 $(i\in V)$;

$N_{out}(i)$: 结点 i 射出边的另一端点集合，$N_{out}(i)=\{j\mid \langle i,j\rangle\in E\}$.

算法6.9 A DISTRIBUTED ALGORITHM FOR SINGLE SHORTEST PATH

输入: 每个结点 $i\in V$ 有 $N_{out}(i)=\{j\mid \langle i,j\rangle\in E\}$，且有权值矩阵中第 i 行 $w(i,j)$,

$j=1, 2, \cdots, n$.

输出: 源 s 到每个结点 i 的最短路径 L_i。若 i 在负环上或从负环可达 i，则 $L_i=-\infty$，$i\in V$。

```
begin
源结点 s 处的算法 (第一阶段)
  初始化
  d_s ← 0;  father(s) ← s;
  send(w(s,i), s) to i, for all i∈N_out(s);
  num_s ← |N_out(s)|;
  upon receiving (l, i) message from i do:
    /* 若检测到负环，开始第二阶段 */
    if l<0 then terminate phase I and start phase II
           else send ACK message to node i
    endif;
  upon receiving an ACK message from j do:
       num_s ← num_s − 1;
       if num_s = 0 then terminate phase I and start phase II endif;
源结点 s 处算法(第二阶段)
  upon receiving (l, i) message from i do:
    if l<0 then send "over −" message to j, for all j∈N_out(s)
            else send "over ?" message to j, for all j∈N_out(s)
    endif;
一般结点 j (j≠s)处算法 (第一阶段)
  初始化
  d_j ← ∞;  father(j) ← j;  num_j ← 0;
  upon receiving (l, i) message from i do:
```

```
    if l < d_j
      then if num_j > 0  then send ACK message to node father(j)  endif;
             father(j) ← i;
             d_j ← l;
             send (d_j + w(j,k), j) message to each k, k∈N_out(j);
             num_j ← num_j + |N_out(j)|;
             if num_j = 0 then send ACK message to father(j) endif
      else send ACK message to node i
    endif;
  upon receiving ACK message from node i do:
    num_j ← num_j - 1;
    if num_j = 0 then send ACK message to father(j) endif;
一般结点j (j≠s)处算法 (第二阶段)
   upon receiving "over-" message  or "over ?" message do:
     if d_j ≠ -∞
        then d_j ← -∞;
               send "over-" message to each i, i ∈N_out(j)
     endif;
   upon receiving "over ?" message do:
       if d_j ≠ -∞ then send "over ?" message to each i, i∈N_out(j) endif;
end.
```

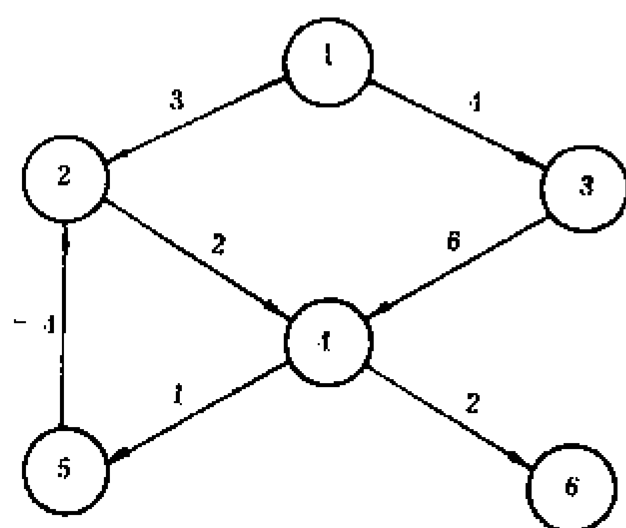


图 6.5 有向加权网络图 $G(V,E)$

下面我们举一个例子来说明算法 6.9 第一阶段的执行情况。图 6.5 是一个加权有向图，其中结点 1 为源点。首先注意，由于发送延迟是任意的，所以网络的计算是不确定的。图 6.5 的四幅快照见表 6.1。说明如下：

快照 1　结点 1 发送一个消息分别给结点 2 和结点 3，它们都还没有收到过消息。

快照 2　结点 2 和结点 3 分别收到消息(3,1)和(4,1)，结点 3 已将(10,3)消息送入结点 4 且结点 4 已收到消息。

快照 3　结点 5 和结点 6 分别从结点 4 收到消息(11,4)和(12,4)。结点 6 送回一个应答消息给结点 4，且结点 4 收到了这个应答消息。接着结点 4 又收到了(5,2)，它送一个应答消息给结点 3。同时向结点 5 和结点 6 发送消息(6,4)和(7,4)，当结点 5 和结点 6 收到消息后，结点 5 送回应答消息给结点 4 且结点 4 已收到这个应答消息。

快照 4 结点 3 送一个应答消息给结点 1．由于结点 3 的 $num_3=0$，结点 5 送回(2,5)到结点 2，这样引起结点 4 送一个应答消息到结点 1．此时结点 1 收到所有的应答消息，它终止算法 6.9 的第一阶段的执行．

表 6.1 算法 6.9 第一阶段执行图 6.5 的快照

快 照	结点 i	1	2	3	4	5	6
1	d_i	0	∞	∞	∞	∞	∞
	father(i)	1	2	3	4	5	6
	num_i	2	0	0	0	0	0
2	d_i	0	3	4	10	∞	∞
	father(i)	1	1	1	3	5	6
	num_i	2	0	1	0	0	0
3	d_i	0	3	4	5	6	7
	father(i)	1	1	1	2	4	4
	num_i	2	1	0	2	0	0
4	d_i	0	2	4	5	6	7
	father(i)	1	5	1	2	4	4
	num_i	0	1	0	2	1	0

6.5.3 算法的正确性证明

首先我们定义有穷结点和无穷结点的概念。一个结点 i，若 $L_i=-\infty$，则 i 为无穷结点；否则，i 是有穷结点．接着我们证明算法 6.9 是正确的。

引理 6.3 从源点 s 到某一结点 j 之间存在一条有穷路径 d_j^*．若第一阶段没有终止，则在某一时刻必有 $d_j \leqslant d_j^*$．

证明 我们对路径上边的数目进行归纳证明，当路径上边的数目为 0 时，引理 6.3 显然成立．现假定引理 6.3 对所有边的数目小于等于 k 的路径都成立．考虑一个从 s 到 j 有 $k+1$ 条边的路径，其中 i 是 j 的父结点。s 到 i 的路径长度为 $d_i^*=d_j^*-w(i,j)$．根据归纳假设，在计算过程中，最终有 $d_i \leqslant d_i^*=d_j^*-w(i,j)$．因此，结点 j 最终收到消息 $(d_i+w(i,j),i)$，它将保证 $d_j \leqslant d_i+w(i,j) \leqslant d_j^*$．根据算法规定，$d_j$ 是下降的．因此在计算过程中 $d_j \leqslant d_j^*$．

引理 6.4 如果第一阶段没有终止，那么在计算的某一时刻，每个无穷结点 j 最终有一个无穷父结点 father(j)，每个有穷结点 j 最终有一个有穷父结点 father(j)，$j \neq 1$．

证明 下式对所有 j，$j \neq 1$ 在任何时候总是成立的．

若 $i=$ father(j)，则 $d_i+w(i,j) \leqslant d_j$，根据最短路径定义 $L_i \leqslant d_i$，若 $i=$ father(j)，则 $L_i+w(i,j) \leqslant d_j$。若 j 是一个无穷结点，则由引理 6.3 可知，最终 d_j 将变得任意小．尤其是，在计算的某个时刻，对每个有穷结点 i 有 $d_i<L_i+w(i,j)$．因此，father(j) 将

是一个无穷结点．由引理 6.3 知，如果第一阶段没有终止，那么每一个有穷结点 i 最终有 $L_i = d_i$，而且 father(i) 将是这条路径上的最后一个中间结点．所以 father(i) 也必须是一个有穷结点．

定理6.5 算法6.9的第一阶段最终会结束．

证明 假定第一阶段永远不终止，那么在算法 6.9 的第一阶段计算的某个时刻，每一个有穷结点 j 有 $d_j = L_j$，因此每个有穷结点从 $d_j = L_j$ 时起不再发送信息．由引理 6.4 有穷结点最终形成一棵根为 s 的有向树，这里 father(j) 是 j 的父结点．在这棵有向树中，叶结点 j ($j \neq s$) 既不可能是任何有穷结点的父结点，也不可能是任何无穷结点的父结点．所以，由引理 6.4，最终有 $\text{num}_j = 0$，结点 j 送一个应答消息给 father(j)．对树的高度进行归纳证明可得：每个有穷结点 j 最终有 $\text{num}_j = 0$。若 s 是一个有穷结点，它将终止第一阶段的计算；若 s 是一个无穷结点，根据引理 6.3，它最终会检测到自己在一个负环中，因而终止第一阶段的执行。因此第一阶段总会结束的。这与前提假定矛盾，定理得证．

定理 6.6 在算法 6.9 的第一阶段终止时，若 j 是一个有穷结点，则 $d_j = L_j$ 且 $\text{num}_j = 0$；若 j 是一个无穷结点，则当且仅当这个时刻，存在某个结点 i，在图 G 中有一条从 s 到 j 的路径，而且这条路径通过 i 且 $\text{num}_i > 0$。

证明 对于一个有穷结点 j，我们定义 $e(j)$ 为从 s 到 j 的最短路径的边数．若存在几条最短路径，则选择其中一条无环且边数最多的路径之后，用归纳法对所有结点 j（其中 $e(j) \leqslant k$，$k = 0, 1, \cdots$）进行归纳证明即可。

对于无穷结点来讲，我们采用反证法证明之，假定每个无穷结点 j，在 s 到 j 路径上的所有中间结点 i，有 $\text{num}_i = 0$。在算法 6.9 的第一阶段结束时，即使 s 没有终止第一阶段的计算，j 也绝对没有收到有关的路径长度的消息，因而 d_j 将不会变小，这就与引理 6.3 相矛盾．

定理 6.7 算法 6.9 的第二阶段将结束，且在那时每个结点 j 有 $d_j = L_j$。

证明 第二阶段将终止。因为每个结点至多收到两个消息，即收到 ″over ?″ 消息之后接着收到 ″over $-$″ 消息。没有有穷结点收到 ″over $-$″ 消息，因为在从 s 到一个有穷结点的路径上没有无穷结点．所以在第二阶段，有穷结点 j 的 d_j 保持不变．根据定理 6.6，在第二阶段开始时 $d_j = L_j$．对一个无穷结点来讲，在第一阶段结束时，从 s 到无穷结点 j 之间存在一条经过结点 i 的最短路径，而且 $\text{num}_i > 0$．其结果是：一旦结点 i 收到任何第二阶段的消息，它将传送一个 ″over $-$″ 消息，最终 $d_j = -\infty = L_j$．

6.6 小　　结

本章主要介绍图论上另一个极其重要的问题 —— 最短路径问题的一些并行算法，

具体地讲，基于 SIMD 共享存贮模型，介绍了单源最短路径、以及所有顶点对之间的最短路径的并行算法；基于 Systolic 二维网孔阵列，讨论了所有顶点对之间的最短路径并行算法；基于 MIMD 共享存贮模型，给出了 Moore 算法的并行化实现算法；最后，基于异步通信的分布式计算模型，介绍了一个求单源最短路径的分布式算法。

有关求图的最短路径问题的并行算法，已经取得了一些成果，参见表 6.2 和 6.3。但

表 6.2 单源最短路径问题的并行算法

时 间	作 者	模 型	时间复杂性	处理器或通讯复杂性	备 注
1982	Chen[20]	MIMD	$O(dn^2)$	$O(n)$	
1985	Paige 和 Kruskal[1]	SIMD–EREW PRAM	$O(nm/p+n\log p)$	$O(p)$	
		SIMD–CRCW PRAM	$O(nm/p+n\log\log p)$	$O(p)$	
1987	Lakhani[21]	MIMD	$O(n^2)$	$O(n)$	
1989	Lakshmanan 等人[19]	分布式	$O(n)$	$O(mn)$	

说明：n 为图的顶点数；m 为边数；p 为处理器数；d 为图的直径。

表 6.3 所有顶点对之间的最短路径并行算法

时 间	作 者	模 型	时间复杂性	处理器复杂性	备 注
1981	Dekel 等人[13]	洗牌及超立方	$O(\log^2 n)$	$O(n^3)$	
1982	Reif 和 Spirakis[10]	SIMD–CREW PRAM	$O(\log n\log\log n)$	$O(n^3)$	期望时间
		SIMD–CRCW PRAM	$O(\log\log n)$	$O(n^3)$	期望时间
1984	Frieze 和 Rudolph[11]	SIMD–CRCW PRAM	$O(\log n)$	$O(n^3)$	
1986	Sinha 等人[14]	Systolic 网孔	$O(n\log n)$	$O(n^2)$	

说明：n 是图的顶点个数。

基本上都是基于 Dijkstra 算法及 Floyd 算法，后者又称为传递闭包方法，其实后者的并行化实质上是矩阵乘法的并行化。基于 SIMD–EREW PRAM 和 SIMD–CRCW PRAM 两种计算模型，Paige 等人[1]使用 p 个处理器，给出了时间复杂性分

别为 $O(nm/p+n\log p)$ 和 $O(nm/p+n\log\log p)$ 的单源最短路径算法。Reif 等人[10] 注意到一个含 n 个顶点的随机图 (Random Graph) 平均直径为 $O(\log n)$，求所有顶点对的矩阵乘法，平均只需执行 $O(\log\log n)$ 次，这样，在 SIMD－CREW PRAM 和 SIMD－CRCW PRAM 上，求所有顶点对之间的最短路径，分别需 $O(\log n\log\log n)$ 期望时间、$O(n^3)$ 处理器和 $O(\log\log n)$ 期望时间、$O(n^3)$ 处理器。Frieze 等人[11] 基于 SIMD－CRCW PRAM 计算模型，给出了一个随机求最小值的并行算法，这个算法在使用 $O(n)$ 处理器时，几乎可在 $O(1)$ 时间完成。运用这一结果，他们给出了一个几乎仅需 $O(\log n)$ 时间就可求所有顶点对之间的最短路径的并行算法。算法的期望时间复杂性为 $O(\log\log n)$，处理器复杂性为 $O(n^3)$。基于二维细胞自动机，Levitt 等人[12] 曾给出一个 $O(n)$ 时间、$O(n^2)$ 处理器（细胞）的所有顶点对之间的最短路径算法。Dekel 等人[13] 基于洗牌及超立方这两种互连网络模型，给出了时间复杂性为 $O(\log^2 n)$、处理器复杂性为 $O(n^3)$ 的所有顶点对之间的最短路径算法。Sinha 等人[14] 在二维 Systolic 网孔上，实现了一个所有顶点对之间的最短路径并行算法，这一算法的时间复杂性为 $O(n\log n)$，处理器复杂性为 $O(n^2)$。此外，还有许多其它的最短路径并行算法，在这里不再叙述它们了。有兴趣的读者可查阅本章末的参考文献，进一步了解那些并行算法。

关于最短路径问题的分布式算法，已有众多学者进行研究。这是因为它在通信网络的选路策略中占有举足轻重的地位。对于单源最短路径问题的分布式算法，由于存在负环，Chandy 等人[17] 的算法的复杂性，不是令人满意的，有时甚至呈指数增长。为此，Awerbuch[18] 使用同步器对 Chandy 算法进行了改进。Lakshmanan 等人[19] 利用同步器的结果，得到了一个通信复杂性为 $O(mn)$、时间复杂性为 $O(n)$ 的改进算法。

在 MIMD 机器模型上，Chen[20] 曾给出一个 $O(dn^2)$ 时间、$O(n)$ 处理机的单源最短路径问题的分布式算法。其后，Lakhai[21] 改进了 Chen 的算法，使得在处理器数不变的前提下，时间复杂性降到 $O(n^2)$。最近，Jenq 等人[22] 在 MIMD 超立方互连网络上，实现了所有顶点对之间的最短路径算法。有兴趣的读者可参阅有关文献。

参 考 文 献

[1] Paige R C, Kruskal C P, Parallel Algorthms for Shortest Path Problems, 1985 *Intern. Conf. on Parallel Processing*, 14－19

[2] Dijkstra E W. A Note on Two Problems in Connexion with Graph, *Numererische Mathematik*, 1, 1959, 269－271

[3] Shiloach Y, Vishkin U. Finding the Maximum, Merging and Sorting on a Parallel Computer Model, *J. Algorithms*, 2(1), 1981, 88－102

[4] Floyd R W. Algorithm 97: Shortest Path, *Commu. ACM*, 5(6), 1962, 345

[5] Savage C. Parallel Algorithms for Graph Theoretic Problems, *Ph.D diss., Univ. of Illinois, Urbana,*

1977

[6] Kucera L. Parallel Computation and Conflicts in Memory Access, *Inform. Proc. Lett.*, 14(2), 1982, 93–96

[7] Guibas L J, Kung H T, Thompson C D. Direct VLSI Implementation of Combinatorial Problems, *In Proceedings of the Conf. on VLSI: Architecture, Design, Fabrication*, Calif. Inst. of Tech., Pasadena, 1979, 509–525

[8] Deo N, Pang C Y, Lord R E. Two Parallel Algorithms for Shortest Path Problems, *In Proc.* 1980 *intern. Conf. on Parallel Processing*, 1980, 244–253

[9] Moore E F. The Shortest Path Through a Maze, *in Proc. Intern. Sympo. on Theory of Switching*, 2, 1959, 285–292

[10] Reif J H, Spirakis J. The Expected Time Complexity of Parallel Graph and Digraph Algorithms, TR–11–82, Aikin Computation Lab., Harvard Univ., 1982

[11] Frieze A, Rudolph L. A Parallel Algorithm for all Pairs Shortest Paths in a Random Graph, *Proc. of the 22nd Allerton Conf.*, Univ. of Illinois, 1984, 663–670

[12] Levitt K N, Kantz W T. Cellular Arrays for the Solution of Graph Problems, *Commu. ACM*, 15(9), 1972, 789–801

[13] Dekel E, Nassimi D, Sahni S. Parallel Matrix and Graph Algorithms, *SIAM J. Comput.*, 10(4), 1981, 657–675

[14] Sinha B P, Bhattacharya B B, Ghose S, Srimani P K. A Parallel Algorithm to Compute the Shortest Paths and Diameter of a Graph and Its VLSI Implementation, *IEEE Trans. Computer*, C–35(1), 1986, 1000–1004

[15] Quinn M J, Yoo Y B. Data Structures for the Efficient Solution of Graph Theoretic Problems on Tightly–Coupled MIMD Computers, *in Proc. Intern. Conf. on Parallel Processing*, 1984, 431–438

[16] Quinn M J. *Designing Efficient Algorithms for Parallel Computers*, McGraw–Hill Book Company, 1987

[17] Chandy K M, Misra J. Distributed Computation on Graphs: Shortest Path Algorithms, *Commu. ACM*, 25(11), 1982, 833–837

[18] Awerbuch B. Complexity of Network Synchronization, *JACM*, 32(4), 1985, 804–823

[19] Lakshmanan K B, Thulasiraman K, Comeau U A. An Efficient Distributed Protocol for Finding Shortest Paths in Networks with Negative Weights, *IEEE Trans. Soft. Engine.*, 15(5), 1989, 639–644

[20] Chen C C. A Distributed Algorithm for Shortest Paths, *IEEE Trans. Comput.*, C–31(9), 1982, 898–899

[21] Lakhani G D. An improved Distributed Algorithm for Shortest Path Problems, *IEEE Trans. Comput.*, C–33(9), 1984, 855–857

[22] Jenq J F, Sahni S. All Pairs Shortest Paths on a Hyper–cube Multiprocessor, *In Proc. Intern. Conf. on Parallel Processing*, 1987, 713–716

第七章　矩阵乘法及其在图论算法中的应用

同排序和搜索一样，矩阵乘法在数值计算和非数值计算中扮演着十分重要的角色。本章着重介绍互连网络上的矩阵乘法，然后介绍其在图论算法中的一些应用。

7.1　矩阵乘法的一个简单并行算法

假定计算模型是 SIMD－CREW PRAM，矩阵 $A=(a_{ij})_{n\times n}$，$B=(b_{ij})_{n\times n}$，A 与 B 的乘积矩阵 $C=A\times B=(c_{ij})_{n\times n}(0\leqslant i,j<n)$。按照乘法定义，矩阵 C 定义为：

$$c_{ij}=\sum_{k=0}^{n-1}a_{ik}*b_{kj}$$

由此，我们可直接给出一个并行化算法。算法的形式化描述如下：

算法7.1　PARALLEL MATRIX MULTIPLICATION

输入：矩阵A和B，A、B均为$n\times n$矩阵；

输出：矩阵$C=A\times B$，C是$n\times n$矩阵。

begin

(1) **for** each i, j, k: $0\leqslant i, j, k<n$ **pardo**

(2)　　temp$(i, j, k)\leftarrow a(i, k)*b(k, j)$；　/ * 计算两个对应项的积 * /

(3)　　$c(i, j)\leftarrow\sum_{k=0}^{n-1}$temp$(i, j, k)$

　endfor

end .

定理 7.1　在 SIMD－CREW PRAM 上，计算两个 n 阶矩阵的乘积，算法 7.1 需 $O(\log n)$ 时间、$O(n^3)$ 处理器。

证明　由算法 7.1 可知，算法的第 (2) 步需 $O(1)$ 时间、$O(n^3)$ 处理器；第 (3) 步需 $O(\log n)$ 时间、$O(n^3)$ 处理器。因此整个算法需 $O(\log n)$ 时间、$O(n^3)$ 处理器。

推论 7.1　在 SIMD－CREW PRAM 上，计算两个 n 阶矩阵的乘积，可以使用 $O(n^3/\log n)$ 处理器在 $O(\log n)$ 时间内完成[2]。

证明　在算法 7.1 中，对每个处理器不是每次分配一个元素，而是分配至多

为 $\lceil \log n \rceil$ 个元素，则算法的第 (2) 步需 $O(\log n)$ 时间、$O(n^3/\log n)$ 处理器；第 (3) 步需 $O(\log n)$ 时间，$O(n^3/\log n)$ 处理器．因此整个算法只需 $O(\log n)$ 时间，$O(n^3/\log n)$ 处理器．

7.2 二维网孔上矩阵乘法的下界

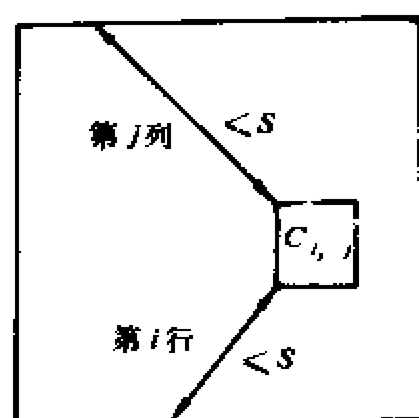
Gentleman[3] 曾经证明：在二维网孔上两个 n 阶矩阵相乘需要 $\Omega(n)$ 数据选路步 (Data Routing Steps)。在介绍这个下界之前，我们先引入几个引理．假定在某种计算模型上，已知一个数据在其中一个处理器中．令 $\sigma(k)$ 表示经过 k 或更少的数据选路步后，数据被广播到的处理器数目的最大值。例如，在二维网孔上，$\sigma(0)=1$，$\sigma(1)=5$，$\sigma(2)=13$，一般情况下，$\sigma(k)=2k^2+k+1$．

引理 7.1 在二维网孔上，两个 n 阶矩阵 A 和 B 进行相乘，假设 A 与 B 的每个元素都仅存放在一个处理器中，且没有处理器含有矩阵 A 或 B 的多个元素．同时还不考虑数据广播设施。则 A 与 B 的乘积矩阵 C 至少需 s 个数据选路步，这里 $\sigma(2s)\geqslant n^2$．

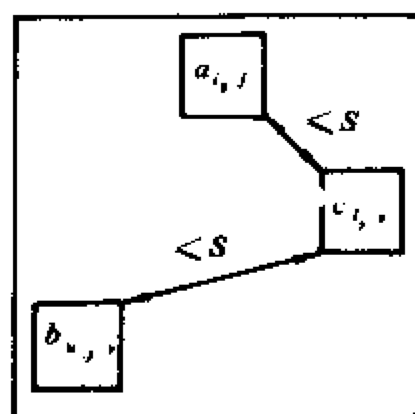
证明 考虑乘积矩阵 C 的任一元素 c_{ij}，它是矩阵 A 的第 i 行元素同矩阵 B 的第 j 列元素的内积．从存贮这些元素的处理器到存贮元素 c_{ij} 的处理器之间必然存在一条路径．令 s 是最长的这样一条路径．如图 7.1(a) 所示．换句话说，创建乘积矩阵至少需 s 个数据选路步。

事实上，注意到这些路径也可用于定义任一元素 b_{uv} 到每个元素 a_{ij} 长度至多为 $2s$ 的路径 ($1\leqslant i,j\leqslant n$)。这是因为从 b_{uv} 存在一条到存贮 c_{iv} 处理器的路径，且这条路径长度至多为 s(如图 7.1(b))。因此存在一条从任一元素 b_{uv} 到每个元素 a_{ij} 的路径，此路径长至多为 $2s$．类似地，从任意一个元素 a_{uv} 到每个 b_{ij} 之间存在一条长度至多为 $2s$ 的路径 ($1\leqslant i,j\leqslant n$)．

A 有 n^2 个元素分别存于唯一的处理器中，由于从存贮元素 b_{uv} 的处理器到存贮 A 元素的处理器路径的长度至多为 $2s$，根据 σ 定义，$\sigma(2s)\geqslant n^2$．



(a) 到 c_{ij} 的路径小于等于 s



(b) 从元素 a_{ij} 到 b_{uv} 不超过 $2s$ 个数据选路步

图 7.1 引理 7.1 的证明示例

定理 7.2 在二维网孔上，两个 n 阶矩阵相乘需 $\Omega(n)$ 个数据选路步，或对很大的 n 来讲，大约要 $s \geqslant 0.35n$ 个数据选路步。

证明 由引理 7.1 可知，$\sigma(2s) \geqslant n^2$。又在二维网孔上 $\sigma(s)=2s^2+2s+1$。即 $\sigma(2s)=2(2s)^2+2(2s)+1$，因此可得：

$$8s^2+4s+1 \geqslant n^2$$

即 $$s^2+s/2+1/8 \geqslant n^2/8$$

$$(s+1/4)^2+1/16 \geqslant n^2/8$$

故 $$s \geqslant \sqrt{n^2/2-1/4}/2-1/4$$

7.3 二维网孔上的矩阵乘法算法

假定在 n^2 个处理器的网孔上，网孔边界有对应的线路（硬线）相连，那么在这种计算模型上设计一个 $O(n)$ 时间的 n 阶矩阵乘法算法是一件比较容易的事。

设矩阵 A、B 及其乘积 C 的元素都存放在对应编号的处理器中，即编号为 (i,j) 的处理器的 $A(i,j)$、$B(i,j)$ 及 $C(i,j)$ 分别存贮 A、B 及 C 对应的元素 $(0 \leqslant i,j<n)$。算法初始化时，$A(i,j) \leftarrow a_{ij}$，$B(i,j) \leftarrow b_{ij}$。算法由两步完成：第一步是 A 与 B 的值对准，结果 $A(i,j) * B(i,j)$ 将是 $C(i,j)$ 的一项（注意：$C(i,j)=\sum_{k=0}^{n-1} a_{ik} * b_{kj}$，由 n 项组成）。所谓对准是指网孔的第 i 行上矩阵 A 的所有元素向左循环移动位置 i 次，同时第 j 列上矩阵 B 的所有元素向上循环移动位置 j 次。完成上述动作后，编号为 (i,j) 的处理器 $A(i,j)$ 的元素为 $a_{i,(j+i) \bmod n}$，$B(i,j)$ 的元素为 $b_{(i+j) \bmod n, j}$，则 $A(i,j) * B(i,j)$ 是 $C(i,j)$ 的一个有效项。第二步是网孔上 A 的每行元素向左移动一个位置，B 的每列元素向上移动一个位置，这时编号为 (i,j) 的处理器中相应元素的乘积 $A(i,j) * B(i,j)$ 是 $C(i,j)$ 的一个新项。作 n 次这样的移动，则得到了 $C(i,j)$ 所有各项。应该注意的是：每当形成 $C(i,j)$ 新的一项时，就把这项加入到 $C(i,j)$ 的部分和上，结果 $C(i,j)$ 是 A 的第 i 行和 B 的第 j 列的内积。

二维网孔上矩阵乘法算法的形式化描述如下：

算法7.2 MATRIX MULTIPLICATION ON MESH ARRAY

输入：每个编号为 (i,j) 的处理器存贮有矩阵A及矩阵B的元素a_{ij}及$b_{ij}$$(0 \leqslant i,j<n)$；

输出：每个编号为 (i,j) 的处理器存贮乘积矩阵C的元素$c_{ij}$$(0 \leqslant i,j<n)$.

```
begin
(1) for l←0 to n－1 do      /＊A和B的值对准＊/
(2)     for each i, j : 0≤i, j<n pardo
(3)         if i>l then A(i, j)⇐A(i, (j+1) mod n) endif;
```

```
(4)          if j>l then B(i, j)⇐B((i+1) mod n, j) endif;
          endfor
       endfor;
(5) for each i, j : 0≤i, j<n pardo      /* C赋初值 */
(6)        C(i, j)←A(i, j)*B(i,j)
       endfor;
(7) for l←1 to n−1 do      /* 计算C的其它项 */
(8)      for each i, j : 0≤i, j<n pardo
(9)          A(i, j)⇐A(i, (j+1) mod n);
(10)         B(i, j)⇐B((i+1) mod n, j);
(11)         C(i, j)←C(i, j)+A(i, j)*B(i, j)
         endfor
     endfor
end.
```

算法中"⇐"符号表示将一个处理器中的数据传递到另一个处理器中去的操作。

定理 7.3 在边界有硬线连接的二维网孔上，计算两个 n 阶矩阵乘积的算法 7.2 需 $O(n)$ 时间、$O(n^2)$ 处理器。

证明 算法 7.2 的第 (1) 到 (4) 步需 $O(n)$ 时间、$O(n^2)$ 处理器；第 (5) 到 (6) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (7) 到 (11) 步需 $O(n)$ 时间、$O(n^2)$ 处理器、因此，整个算法需 $O(n)$ 时间、$O(n^2)$ 处理器。

在上述算法中，假定二维网孔计算模型边界有硬线连接。即编号为 $(i,0)$ 的处理器同编号为 $(i, n-1)$ 的处理器有线互连，编号为 $(0,i)$ 的处理器同编号为 $(n-1,i)$ 的处理器也有线互连 $(0 \leqslant i < n)$。事实上，边界约束条件可以放松。Dekel 等人已证明了在边界没有硬线连接的情况下，二维网孔上矩阵乘法仍然只需 $O(n)$ 时间、$O(n^2)$ 处理器[1]。

7.4 超立方上的矩阵乘法算法

Dekel 等人在超立方等互连网络模型上实现了矩阵乘法。他们算法的关键是给出了一种巧妙的**数据选路策略** (Data Routing Strategy)。在给出算法之前，先引入一些记号和术语。

令一个整数 $i > 0$ 的二进制表示的第 r 位为 i_r，$0 \leqslant r \leqslant \log n$，约定 $n = 2^q$。i 的二进制表示为 $i_{q-1}i_{q-2}\cdots i_0$。$i^{(b)}$ 表示一个二进制数：$i_{q-1}i_{q-2}\cdots i_{b+1}\bar{i}_b i_{b-1}\cdots i_0$，其中 $\bar{i}_b = 1 - i_b$。

所谓**超立方计算模型**是由 $n^3 = 2^{3q}$ 个处理器组成的网络。从原理上讲，我们将这 n^3

个处理器想象成 $n\times n\times n$ 的晶格排列，然后按超立方互连网络规则处理器互连起来。处理器采用行主方式编号。也就是说，位于阵列 (i, j, k) 位置的处理器编号是 $i*n^2+j*n+k$，若此编号的二进制表示为 $r_{3q-1}\cdots r_{2q-1}\cdots r_0$，则

$$i=r_{3q-1}\cdots r_{2q},\ j=r_{2q-1}\cdots r_q,\ k=r_{q-1}\cdots r_0\ (0\leqslant i, j, k<n).$$

令 $A(i, j, k)$，$B(i, j, k)$ 及 $C(i, j, k)$ 分别是编号为 $i*n^2+j*n+k$ 的处理器的寄存器。矩阵 A 及矩阵 B 的元素在算法开始前已存放在下述编号的处理器中，$A(0,j,k)=a_{jk}$，$B(0, j, k)=b_{jk}$。算法结束时乘积矩阵 C 的元素存放在 $C(0, j, k)=c_{jk}$ 中，$(0\leqslant i, j, k<n)$。

算法分三步执行：第一步将矩阵 A 与 B 的元素按如下的要求广播到 n^3 个处理器中。广播后位于 (l, j, k) 处理器的寄存器内容分别是：$A(l, j, k)=a_{jl}$，$B(l, j, k)=b_{lk}$；第二步给乘积矩阵 C 赋初值，位于 (l, j, k) 处理器的寄存器内容为 $C(l, j, k)\leftarrow A(l, j, k)*B(l, j, k)$，即对应项的积是 $a_{jl}*b_{lk}$；最后一步计算 C 的其它项的和，即计算 $\sum_{l=0}^{n-1}C(l, j, k)$。这里 $0\leqslant l,\ j,\ k<n$。

超立方上矩阵乘法算法的形式化描述如下：

算法7.3 MATRIX MULTIPLICATION ON HYPERCUBE

输入：编号为 $i*n+j$ 的处理器存放矩阵A及矩阵B的元素a_{ij} 及 $b_{ij}(0\leqslant i, j<n)$;

输出：编号为 $i*n+j$ 的处理器存放矩阵C的元素$c_{ij}(0\leqslant i, j<n)$.

```
begin
/ * (1)至(3)步对数据进行广播，将数据放入相应的处理器中，即初值对准 * /
(1)  for l←3q−1 downto 2q do
        for each j : 0≤j<n³ pardo
            if j_l = 0 then A(j^(l))←A(j) endif;
            if j_l = 0 then B(j^(l))←B(j) endif;
        endfor
     endfor;
(2)  for l←q−1 downto 0 do   / * 将A(i, j, i)的内容拷贝到A(i, j, * )中 * /
        for each j : 0≤j<n³ pardo
            if j_l = j_(2q+l) then A(j^(l))←A(j) endif
        endfor
     endfor;
(3)  for l←2q−1 downto q do     / * 将B(i, i, k)内容拷贝到B(i, * , k)中 * /
        for each j : 0≤j<n³ pardo
```

```
            if j_l = j_{l+q} then B(j^(l)) ⇐ B(j) endif;
        endfor
    endfor;
(4) for each j : 0 ≤ j < n^3 pardo   /* 乘积矩阵初始化 */
        C(j) ← A(j) * B(j)
    endfor;
(5) for l ← 2q to 3q − 1 do
      /* 对各项内容相加，结果放入规定编号的处理器中 */
        for each j: 0 ≤ j < n^3 pardo
            Temp(j) ⇐ C(j^(l));
            C(j) ← C(j) + Temp(j)
        endfor
    endfor
end.
```

注意：$A(i, j, k) = A(i * n^2 + j * n + k)$，$B$，$C$ 也类似 $(0 \leqslant i, j, k < n)$。图 7.2 例示了在 2^3 个处理器组成的超立方上，两个 2 阶矩阵 A 与 B 乘法运算的执行过程.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} -5 & -6 \\ 7 & 8 \end{bmatrix}$$

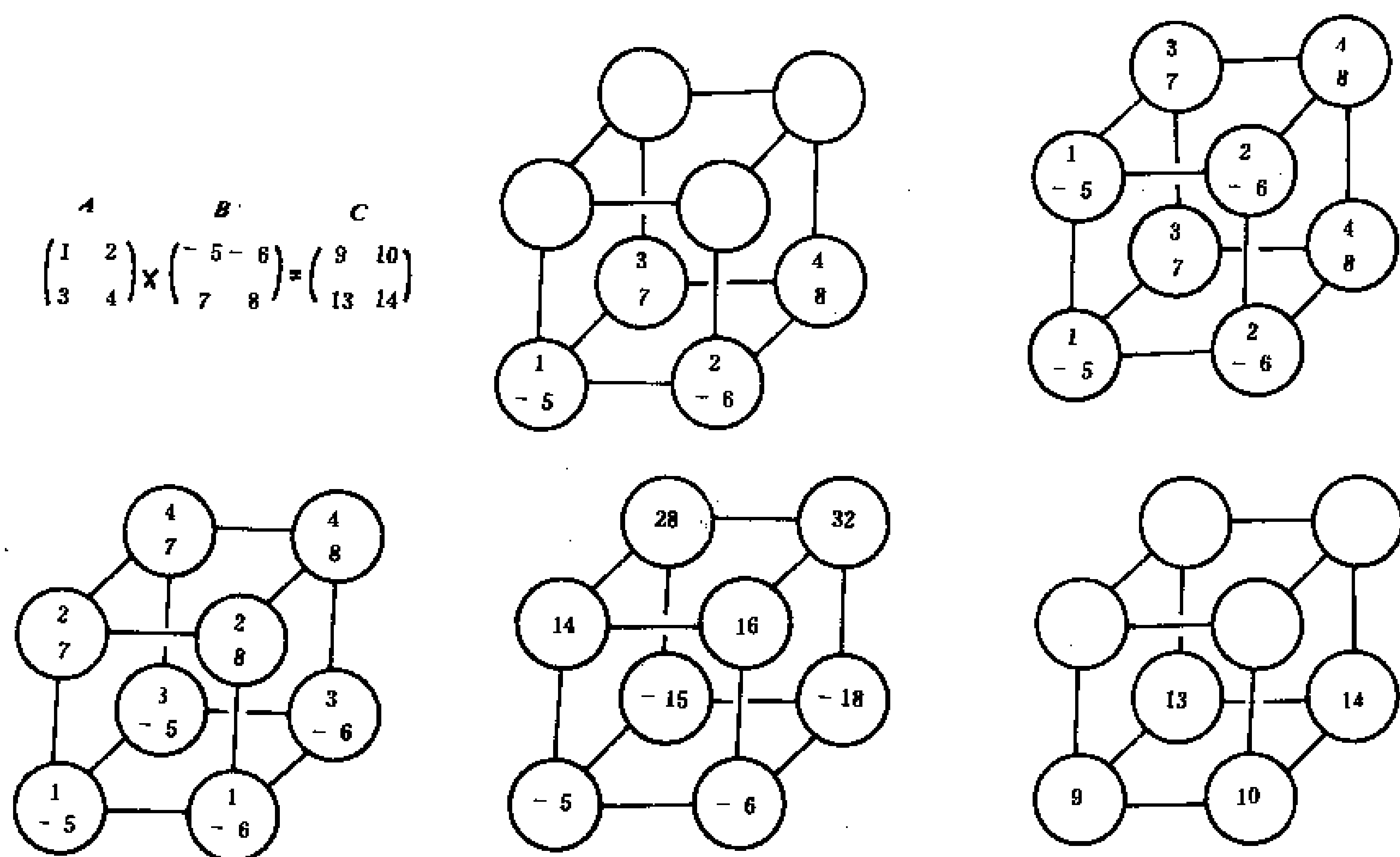


图 7.2 SIMD–超立方模型上的并行矩阵乘法

定理 7.4 在超立方上，计算两个 n 阶矩阵的乘积矩阵的算法 7.3 需 $O(\log n)$ 时间、$O(n^3)$ 处理器。

证明 算法 7.3 的第 (1) 步需 $O(q)$ 时间、$O(n^3)$ 处理器；第 (2) 步需 $O(q)$ 时间、$O(n^3)$ 处理器；第 (3) 步需 $O(q)$ 时间、$O(n^3)$ 处理器；第 (4) 步需 $O(1)$ 时间、$O(n^3)$ 处理器；第 (5) 步需 $O(q)$ 时间、$O(n^3)$ 处理器。因此整个算法需 $5q$ 个数据选路步，且 $n=2^q$，所以算法需 $O(\log n)$ 时间、$O(n^3)$ 处理器。

Dekel 等人还给出了处理器数分别为 n^2，n^2p 及 $p^2(1 \leqslant p \leqslant n)$ 时的超立方上矩阵乘法算法，这些算法的基本原理同本节是一致的，但算法中数据选路更复杂，故不在此介绍。

7.5 洗牌网络上的矩阵乘法算法

上面我们介绍了超立方上的矩阵乘法算法。这儿我们将介绍 Dekel 等人[1] 在洗牌网络上的矩阵乘法算法，算法假定有 n^3 个处理器。矩阵 A 及矩阵 B 以及乘积矩阵 C 的数据在处理器的存放方式同在超立方上的存放完全一致。算法的基本步骤同超立方上算法的步骤也相同。但由于互连拓扑结构的不同，因而算法实现细节也有些不同。

下面一些语句将在算法中用到：

(1) $A(j^{(0)}) \Leftarrow A(j)$，根据 Exchange 函数选路；

(2) $A(\text{Shuffle}(j)) \Leftarrow A(j)$，根据 Shuffle 函数选路；

(3) $A(\text{Unshuffle}(j)) \Leftarrow A(j)$，根据 Unshuffle 函数选路；

这里：$0 \leqslant j < n^3$。数据初始分布仍然是位于 $(0, i, t)$ 位置的处理器的寄存器 $A(0, i, t) = a_{it}$，$B(0, i, t) = b_{it}$，$0 \leqslant i,\ t < n$。

洗牌网络上矩阵乘法算法的形式化描述如下：

算法 7.4 MATRIX MULTIPLICATION ON SHUFFLE – EXCHANGE

输入：编号为 $i*n+j$ 的处理器存贮矩阵A的元素 a_{ij} 及矩阵B的元素 $b_{ij}(0 \leqslant i, j < n)$；

输出：编号为 $i*n+j$ 的处理器存贮矩阵 C 的元素 $c_{ij}(0 \leqslant i, j < n)$。

```
begin
/ * (1)至(3)步的功能是数据广播，将数据放入相应处理器中，即初值对准 * /
(1) for b ← 0 to q − 1 do   / * 复制A, B * /
        for each i : 0 ≤ i < n³ pardo
            A(Shuffle (i)) ⇐ A(i) ;
            B(Shuffle (i)) ⇐ B(i) ;
```

```
            if i_0 = 0 then A(i^(0)) ⇐ A(i) endif;
            if i_0 = 0 then B(i^(0)) ⇐ B(i) endif
        endfor
    endfor;  / * 此刻A(i, t, p) = a_it, B(i, t, p) = b_it * /
(2) for b ← 0 to q − 1 do
        for each i : 0 ≤ i < n^3 pardo
            A(Shuffle (i)) ⇐ A(i)
        endfor
    endfor;  / * 此刻A(i, t, p) = a_pt, B(i, t, p) = b_it * /
(3) for each i: 0 ≤ i < n^3 pardo  / * 乘积矩阵初始化 * /
        C(j) ← A(j) * B(j)
    endfor;  / * 此刻C(i, t, p) = a_pi * b_it * /
(4) for b ← 0 to q − 1 do
        for each i: 0 ≤ i < n^3 pardo  / * 计算乘积矩阵C的各项 * /
            C(Shuffle (i)) ⇐ C(i);
            temp(i) ⇐ C(j^(0));
            C(j) ← C(j) + temp(i)
        end for
    endfor;  / 此时, C(t, p, i) = C_pt, C(t, p, t) = C_pt * /
(5) for b ← 0 to q − 1 do
        for each i: 0 ≤ i < n^3 pardo
            C(Shuffle(i)) ⇐ C(i);
            if (j_0 = 1) and (j_q = 1) then C(j^(0)) ⇐ C(j) end if
        endfor
    endfor;  / C(p, t, 0) = C_pt * /
(6) for b ← 0 to q − 1 do  / * 将结果放入要求的处理器中 * /
        for each i: 0 ≤ i < n^3 pardo
            C(Unshuffle(i)) ⇐ C(i)
        endfor
    endfor       / * C(0, p, t) = C_pt * /
end.
```

定理 7.5 在洗牌网络上，计算两个 n 阶矩阵的乘积矩阵的算法 7.4 需 $O(\log n)$ 时间、$O(n^3)$ 处理器.

证明 通过对算法 7.4 的每步复杂性分析，我们很容易给出整个算法的复杂性. 算法 7.4

的第 (1) 步需 $4q$ 个数据选路步；第 (2) 步需 q 个数据选路步；第 (3) 步需 $O(1)$ 个计算步；第 (4) 步需 $2q$ 个数据选路步；第 (6) 步需 q 个数据选路步。故整个算法需 $O(q)$ 个计算步、$10q$ 个数据选路步，而算法每一步均需 $O(n^3)$ 处理器，故算法需 $O(q)=O(\log n)$ 时间、$O(n^3)$ 处理器。

Dekel 等人还设计了处理器数分别为 n^2，$n^2 p$ 及 $p^2 (1 \leqslant p \leqslant n)$ 的洗牌网络上的矩阵乘法算法。这里就不一一再作介绍了。

7.6 MIMD 共享存贮模型上的矩阵乘法算法

在 MIMD 紧耦合共享存贮模型上，对单机上的矩阵乘法提供了各种并行化机会。为叙述方便，这里给出单机上的矩阵乘法算法。假定矩阵$A=(a_{ij})_{l\times m}$，矩阵$B=(b_{ij})_{m\times n}$，乘积矩阵$C=(c_{ij})_{l\times n}$。单机上矩阵乘法算法为:

算法7.5 MATRIX MULTIPLICATION (SISD)

```
procedure Matrix_Multiple (A, B, C);   /* C = A × B */
  begin
  (1)  for i←0 to l−1 do
     (1.1)    for j←0 to n−1 do
          (1.1.1)  t←0;
          (1.1.2)  for k←0 to m−1 do
                        t←t + a_ik * b_kj
                   endfor;
          (1.1.3)  c_ij←t
              endfor
    endfor
  end.
```

此算法有三个 for 循环可以并行化。这就引出了一个有趣的问题：当存在许多适于并行化的嵌套循环时，应该使哪个循环并行化呢？下面的引理给出了在紧耦合多处理器上设计并行算法的一般指导原则。

引理 7.2 已知一个具有 p 个活动处理器的紧耦合多处理机系统，它们必须经过一个全局信号灯进行同步；且一个时间复杂性在最坏情况下为 $t(n)$ 的任务，在它完成之后需要同步。那么最大可能的加速为 $t(n)/\lceil\sqrt{2t(n)+1/4}+1/2\rceil$ 或者为 $O(\sqrt{t(n)})$。

证明 为了通过全局信号灯同步，每个进程必须对信号灯进行上锁、增值以及去锁操作。不失一般性，假定这些操作仅需 $O(1)$ 时间。因此 p 个处理器同步至少需 $O(p)$ 时间。在同步期间，工作量需 $p(p-1)/2$ 时间（见图 7.3）。当 $p=\lceil\sqrt{2t(n)+1/4}$

$+1/2]$ 时，能完成的工作量至少是 $t(n)$，因此 $O(\sqrt{t(n)})$ 个处理器即可使处理时间达到极小化。增加更多的处理器仅能增加整个执行时间（见图 7.4)。

上述引理 7.2 对并行矩阵乘法有何意义呢？最内层的 for 循环时间复杂性为$O(n)$，若对这个循环进行并行化，可获得的最大加速为$O(\sqrt{n})$。中间的 for 循环时间复杂性为$O(n^2)$，若使这个 for 循环并行化，算法可获得$O(n)$的加速。最后，最外层的 for 循环时间复杂性为$O(n^3)$，将这个 for 循环并行化，可使加速达到$O(n^{1.5})$。当然，象算法的可划分性以及 A 与 B 的元素冲突等其它因素不可能得到这么多的加速。但一般来讲，根据上述的粒度尺寸引理 7.2，当需要给出选择时，总是选择最外层循环进行并行化。

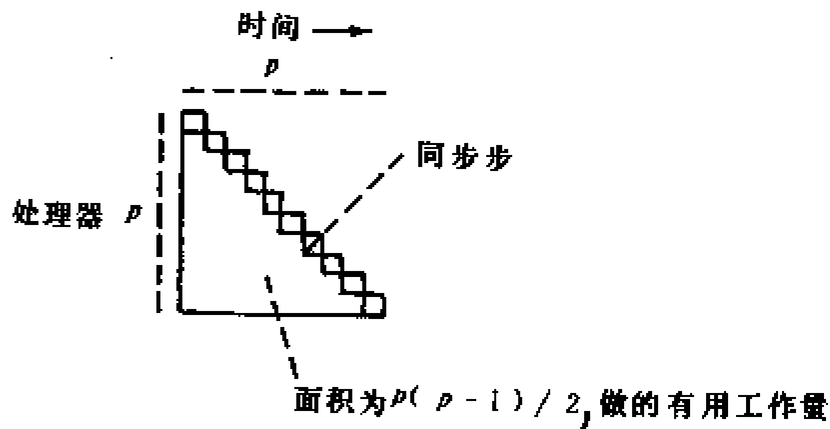


图 7.3 在 MIMD 模型上同步期间的处理

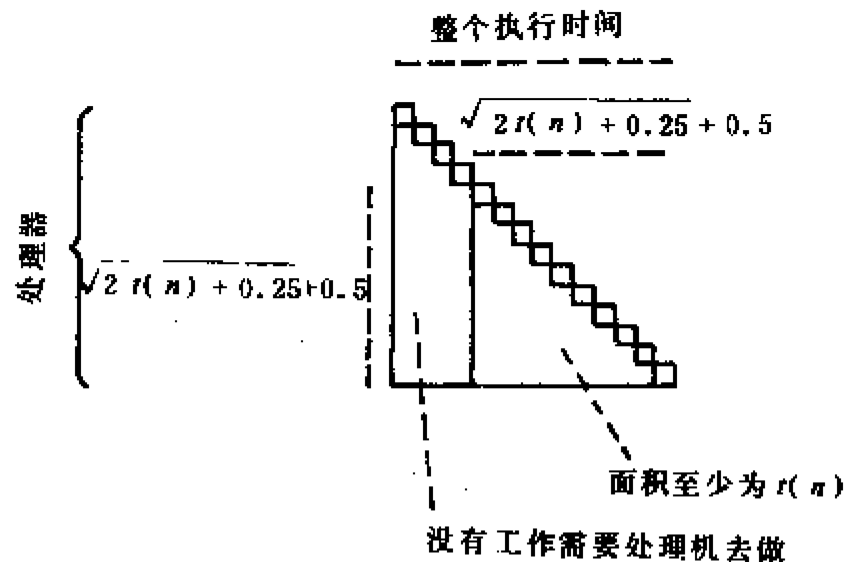


图 7.4 增加处理器可能增加的执行时间

MIMD模型上的矩阵乘法算法的形式化描述如下：

算法7.6 MATRIX MULTIPLICATION ON MIMD MACHINE

输入：矩阵 A 和矩阵 B；

输出：矩阵 $C = A \times B$。

```
begin
(1)  for each l: 1≤l≤p pardo
(2)       for i←l to n step p do
(3)            for j←i to n do
(4)                 t←0;
(5)                 for k←1 to n do
(6)                      t←t+a_ik * b_kj
                    endfor;
(7)                 c_ij←t
               endfor
          endfor
     endfor
end.
```

定理 7.6 在 MIMD 紧耦合共享存贮模型上，计算两个 n 阶矩阵乘积的算法 7.6 需 $O(n^3/p+p)$ 时间、$O(p)$ 个处理器 $(1 \leqslant p \leqslant n)$。

证明 由算法 7.6 可知：每个进程计算了矩阵 C 的 n/p 行。而计算矩阵的一行需 $O(n^2)$ 时间，进程间同步仅一次，因此同步开销是 $O(p)$ 时间。所以整个并行算法需 $O(n^3/p+p)$ 时间、$O(p)$ 处理器 $(1 \leqslant p \leqslant n)$。

若忽略了存贮冲突，我们可以得到近似线性的加速。但忽略内存存取次数合理吗？假定每个处理器到共享存贮单元是等同的，在紧耦合多机上这一假设是安全的，但对松散耦合的多处理器来讲，这一假设是危险的。这是因为在这种计算模型上，存取矩阵的一些元素可能较存取另外一些元素容易得多。在象 C_m^* 这样松散耦合处理器上，将众多的共享存贮单元地址保存在局部单元上是很重要的。上述算法在这种计算模型上不能很好地工作。事实上，一个典型的进程不仅必须存取 A 的 n/p 行，而且它还必须存取 B 的每个元素 n/p。而对提取 B 的每个元素仅作一次加法和乘法运算，这不是一个很好的比率。上述算法在 C_m^* 上实现将产生非常小的加速。

所以我们必须寻找一种划分问题的方法。一种诱人的方法是利用块矩阵乘法。假定 A 和 B 均为 n 阶矩阵，这里 $n=2k$，那么 A 和 B 可被想象是四个 k 阶矩阵的集合。

$$A=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B=\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

已知矩阵A和B已划分成块，那么C定义为：

$$C=\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}=\begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{bmatrix}$$

若我们将进程分配做块矩阵乘法。那么每次提取的矩阵元素做加法和乘法运算的次数也增加。例如，假定有 $p=(n/k)^2$ 个进程，然后在 A 和 B 划分成 p 块 k 阶子矩阵上执行矩阵乘法。每个块矩阵乘法需 $2k^2$ 次存贮存取，k^3 次 加 法 和 k^3 次乘法。若 $k=n/\sqrt{p}$，则新的算法较以前的算法有显著的改进。图 7.5 是将块矩阵乘法应用到并行矩阵乘法上的一个例子。

$$\begin{bmatrix} 1 & 0 & 2 & 3 \\ 4 & -1 & 1 & 5 \\ -2 & -3 & -4 & 2 \\ -1 & 2 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} -1 & 1 & 2 & -3 \\ -5 & -4 & 2 & -2 \\ 3 & -1 & 0 & 2 \\ 1 & 0 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 8 & -1 & 14 & 16 \\ 9 & 7 & 26 & 17 \\ 7 & 14 & -2 & 14 \\ -9 & -9 & 2 & -1 \end{bmatrix}$$

步　骤 1：计算　$C_{ij}=A_{i,1}B_{1,j}$

$$
\begin{matrix}
\begin{bmatrix}1 & 0\\ 4 & -1\end{bmatrix} & \begin{bmatrix}-1 & 1\\ -5 & -4\end{bmatrix} & \begin{bmatrix}1 & 0\\ 4 & -1\end{bmatrix} & \begin{bmatrix}2 & -3\\ 2 & -2\end{bmatrix}\\
\begin{bmatrix}-2 & -3\\ -1 & 2\end{bmatrix} & \begin{bmatrix}-1 & 1\\ -5 & -4\end{bmatrix} & \begin{bmatrix}-2 & -3\\ -1 & 2\end{bmatrix} & \begin{bmatrix}2 & -3\\ 2 & -2\end{bmatrix}
\end{matrix}
=\begin{bmatrix}-1 & 1 & 2 & -3\\ 1 & 8 & 6 & -1\\ 17 & 10 & -10 & 12\\ 0 & -9 & 2 & -1\end{bmatrix}
$$

步　骤 2：计算　$C_{ij}+A_{i,1}B_{2,j}$

$$
\begin{matrix}
\begin{bmatrix}2 & 3\\ 1 & 5\end{bmatrix} & \begin{bmatrix}3 & -1\\ 1 & 0\end{bmatrix} & \begin{bmatrix}2 & 3\\ 1 & 5\end{bmatrix} & \begin{bmatrix}0 & 2\\ 4 & 5\end{bmatrix}\\
\begin{bmatrix}-4 & 2\\ 0 & 0\end{bmatrix} & \begin{bmatrix}3 & -1\\ 1 & 0\end{bmatrix} & \begin{bmatrix}-4 & 2\\ 0 & 0\end{bmatrix} & \begin{bmatrix}0 & 2\\ 4 & 5\end{bmatrix}
\end{matrix}
=\begin{bmatrix}8 & -1 & 14 & 16\\ 9 & 7 & 26 & 17\\ 7 & 14 & -2 & 14\\ -9 & -9 & 2 & -1\end{bmatrix}
$$

图 7.5　并行矩阵乘法的分块矩阵法

7.7　矩阵乘法在图论算法中的应用

7.7.1　计算图的传递闭包

设一个有向图 $G(V,E)$，$|V|=n$，约定 $n=2^q$。图 G 的邻接矩阵为 A，则 A 的传递闭包 $A^*=(A+I)^n$，I 是 n 阶单位矩阵。A^* 的计算可由下式完成。

$$(A+I)^n=((\cdots((A+I)^2)^2\cdots)^2)^2\text{①}$$

因而图 G 的传递闭包 A^* 可通过 $\lceil \log n\rceil$ 次并行矩阵乘法得到。

定理 7.7　在超立方和洗牌网络上，计算 n 阶布尔矩阵的传递闭包需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器。

证明　根据定理 7.4 及定理 7.5，在超立方和洗牌网络上计算两个 n 阶矩阵乘积需 $O(\log n)$ 时间和 $O(n^3)$ 处理器；而计算一个布尔矩阵的传递闭包需进行 $\log n$ 次矩阵乘法。因此在这两种计算模型上计算 n 阶布尔矩阵传递闭包需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器。

7.7.2　计算所有顶点对的最短路径

令有向加权图 $G(V,E)$，$|V|=n$，所有顶点对的最短路径矩阵为 P，这里 $P(i,j)$ 表示顶点 i 到顶点 j 的最短路径长度。令 $P^{(k)}(i,j)$ 表示从 i 到 j 至多经过 k 个中间顶点的一条最短路径。显然，$P(i,j)=P^{(n)}(i,j)$。若 $<i,j>\in E$，则 $P^{(0)}(i,j)=w(i,j)$，其中 W 是 G 的加权邻接矩阵；若 $<i,j>\notin E$，则 $P(i,j)=\infty$，$P^{(0)}(i,i)=0$。不难看出，

① $\lceil \log n\rceil$个2

$$P^{(k)}(i,j)=\min_{0\leqslant l<n}\{P^{(k/2)}(i,l)+P^{(k/2)}(l,j)\}$$

其中，$k=0$，2，4，8，16，…，n，$0\leqslant i,j<n$。因此矩阵 $P^{(n)}$ 可由矩阵 $P^{(0)}$ 开始执行矩阵乘法运算获得，即计算序列 $P^{(0)}$，$P^{(2)}$，$P^{(4)}$，…，$P^{(n/2)}$，$P^{(n)}$。$P^{(k)}$ 的计算将矩阵的乘法算符"*"换成"+"，矩阵的加法算符"+"换成"min"即可。$P^{(n)}$ 需经 $\log n$ 次矩阵乘法运算才能获得。为此得:

定理 7.8 在超立方和洗牌网络上，计算一个有向加权图 $G(V,E)$，$|V|=n$ 的所有顶点对的最短路径需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器。

证明 同定理 7.7 一致，故不再赘述。

推论 7.2 在超过立方和洗牌网络上，计算一个有向加权图 $G(V,E)$，$|V|=n$ 的所有顶点对的最短路径只需 $O(\log d\cdot\log n)$ 时间、$O(n^3)$ 处理器，这里 d 是图的直径。

证明 在超立方或洗牌网络上执行矩阵乘法需 $O(\log n)$ 时间、$O(n^3)$ 处理器。又图的直径为 d，故最短路径矩阵 $P=P^{(d)}$。所以经过 $\lceil\log d\rceil$ 次矩阵乘法运算，就可算出 P，故整个算法需 $O(\log d\cdot\log n)$ 时间、$O(n^3)$ 处理器。

7.7.3 计算图的中值和中值长度

令 $G(V,E)$，是一个有向加权图，$|V|=n$。令 $d(i,j)$ 表示从顶点 i 到顶点 j 的最短路径长度，令 $h(i)$ 表示顶点 i 的权值 $(0\leqslant i,j<n)$。顶点 $v\in V$ 是图的**加权中值**当且仅当 $\sum_{j=0}^{n-1}h(j)d(v,j)\leqslant\sum_{j=0}^{n-1}h(j)d(k,j)$，$0\leqslant k<n$。当任意的 $j\in V$ 均有 $h(j)=1$ 时，顶点 v 称为图的**中值**。$\sum_{j=0}^{n-1}h(j)d(v,j)$ 称为**加权中值长度**。

由上述中值及加权中值长度定义可知，此问题算法的关键是求图的所以顶点对的最短路径。前面我们已给出了最短路径算法：对所有 $k\in V$，求和 $\sum_{j=0}^{n-1}h(j)d(k,j)$ $=\sum_{j=0}^{n-1}h(j)A(k,j)$，在 n^3 处理器的超立方和洗牌网络上只需 $O(\log n)$ 时间就可完成。然后求这些和的最小值也可在相同时间内完成。

定理 7.9 在超立方和洗牌网络上，计算一个有向加权图 $G(V,E)$，$|V|=n$ 的中值及中值长度需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器。

证明 见本小节的叙述。

Dekel 等人还给出矩阵乘法在其它图论算法中的应用，如求图的半径，直径和中心，

求图的最短路径生成树、宽度优先生成树，求 AOE 网的拓扑排序以及求关键路径等。所有这些问题算法，在超立方和洗牌网络上实现只需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器。

7.8 小　　结

本章介绍了许多图论算法都用到的一个重要子过程 —— 矩阵乘法算法。主要内容有：首先基于 SIMD – CREW PRAM，叙述了一个简单的矩阵乘法并行算法；其次基于二维网孔，超立方以及洗牌网络，分别介绍了在这些网络模型上的矩阵乘法算法；另外还给出了基于 MIMD 共享存贮模型上的矩阵乘法算法；最后给出了矩阵乘法在一些图论算法中（如计算传递闭包，求最短路径以及计算中值及中值长度）的典型应用。

本章介绍的所有矩阵乘法算法，都是最简单的单机上算法的并行化。Chandra[5] 曾对著名的 Strassen 矩阵乘法算法进行了并行化，在 SIMD – CREW PRAM 上，他获得了一个 $(n^{\log 7}/p)$ 时间、$O(p)$ 处理器的算法，$p \leqslant n^{\log 7}/\log n$。Ramakrishnan 等人在一维线性阵列上给出了 $O(n^3/p)$ 时间、$O(p)$ 处理器算法[6]。Hwang 等人设计了 VLSI 块矩阵乘法算法，用以说明当处理器数目小于矩阵元素个数时怎样执行矩阵乘法[7]。Savage 基于 SIMD – CREW PRAM 曾给出一个 $O(\log n)$ 时间、$O(n^3/\log n)$ 处理器算法[2]。Agerwala 等人曾获得一个有效的布尔矩阵乘法算法，他们的算法是四个俄国人算法的并行实现[8]。Van Scoy 和 Flynn 等人基于二维网孔，分别给出了 $O(n)$ 时间、$O(n^2)$ 处理器算法[9,10]。Cannon 曾基于二维心动网孔阵列给出了 $O(n)$ 时间、$O(n^2)$ 处理器算法，但他的网孔假定边界是硬线连接的[11]。基于 VLSI 计算模型，Leighton 在树网上给出了一个 $O(n\log n)$ 时间、$O(n^2)$ 处理器算法[12]。Leiserson 等人在心动阵列上给出了矩阵乘法算法[13]。Ullman 则较系统地介绍了这方面的工作[14]。最近，梁维发和陈国良在树网模型上建议了一个新的矩阵乘法算法，他们的算法需 $O(n\log n)$ 时间、$O(n^2)$ 处理器[17]。利用这一算法，可有效地模拟二维网孔模型上的一类算法。

前面我们列举了矩阵乘法在图论算法中应用的例子。实际上远不止这些，如 Kim 等人基于 SIMD – CREW PRAM，利用矩阵乘法分别给出了一般图的宽度优先搜索算法以及无环有向图的深度优先搜索算法，这些算法需 $O(\log d \cdot \log n)$ 时间、$O(n^3/\log n)$ 处理器[15]。Kim 等人还利用矩阵乘法，在 SIMD – CREW PRAM 上，对二分图最大匹配问题给出了一个 $O(n\log n\log\log n)$ 时间、$O(n^3/\log n)$ 处理器算法[16]。矩阵乘法在图论算法

中应用非常广泛，在以后各章节将再作详细介绍。

参 考 文 献

[1] Dekel E, Nassimi D, Sahni S. Parallel Matrix and Graph Algorithms, *SIAM J. Comput.*, **10**(4),1981,657–675

[2] Savage S. Parallel Algorithms for Graph Theoretic Problems, Ph.D diss., Univ. of Illinois, Urbana, 1977

[3] Gentleman W M. Some Complexity Results for Matrix Computations on Parallel Computers, *J. ACM*, **25**(1), 1978, 112–115

[4] Quinn M J. *Designing Efficient Algorithms for Parallel Computers*, McGraw–Hill Book Company, 1987

[5] Chandra A. Maximal Parallelism in Matrix Multiplication, IBM Tech. Rept. RC6193, 1976

[6] Ramakrishnan I V, Varman P J. Modular Matrix Multiplication on a Linear Array, *IEEE Trans. Computer*, C–**33**(11), 1984, 952–958

[7] Hwang K, Cheng Y H. Partitioned Matrix Algorithms for VLSI Arthmetic System, *IEEE Trans. Computer*, C–**31**(12), 1982, 1215–1224

[8] Agerwala T, Lint B. Communication in Parallel Algorithms for Boolean Matrix Multiplication, *Proc. Intern. Conf. on Parallel Processing*, 1978, 146–153

[9] Van Scoy F. Parallel Algorithms in Cellular Spaces, Ph.D diss., Univ. of Virginia, 1976

[10] Flynn M, Kosaraju R. Processes and Their Interactions, *Kybernetics*, **5**, 1976, 159–163

[11] Cannon L E. A Cellular Computer to Implement the Kalman Fiter Algorithm, Ph.D diss, Montanna State Univ. 1969

[12] Leighton F T. New Lower Bound Techniques for VLSI, *In Proc 22nd Annu Sympo. on FOCS, NY*, 1981, 1–12

[13] Leiserson C E. *Area–Efficient* VLSI *Computation*, MIT Press, MA, 1983

[14] Ullman J D. *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984

[15] Kim T, Chwa K Y. Parallel Algorithms for a Depth First Search and a Breadth First Search, *Intern. J. Computer Math.*, **19**, 1986, 39–54

[16] Kim T, Chwa K Y. An O($n$log$n$loglog$n$) Parallel Maximum Matching Algorithm for Bipartite Graphs, *Inform. Proc. Lett.*, **24**, 1987, 15–17

[17] 梁维发，陈国良．树网结构上的并行图论算法，计算机研究与发展．**28** (1)，1991，1–8

第八章　基本回路、关节点、双连通分支和桥的并行算法

在图论及其应用中，无向图的基本回路、关节点、双连通分支和桥，是常常出现的问题。这些问题也都属于图论应用的基本问题。在并行环境中，如何快速地计算图的这些基本性质，不仅具有重要的理论意义，而且具有很大的应用价值。这一章我们来介绍求无向图的基本回路、关节点、双连通分支和桥的并行算法。

在第五章的第 5.4 节，我们已经引进了逆树、最低公共祖先等基本概念和记号，这一章将继续沿用它们。尚不熟悉的读者，请查阅该节介绍的有关内容。

8.1　逆树的一些基本性质

为了方便后面的分析，首先我们介绍一些基本的引理。

引理 8.1　在 SIMD－CREW PRAM 上，(1) 已知一个由逆树组成的有向森林 $T(V,E')$，其中有向森林函数 F 为 $F: V \to V$，若要计算全部 $F^k(i)$，$i \in V$，$0 \leqslant k < n$，且 $F^k(i)$ 存贮在二维数组 F^+ 的 $F^+(i,k)$ 单元中，则需 $O(n/K+\log n)$ 时间、$O(nK)$ 处理器；(2) 已知所有 $F^k(i)$，$i \in V$，$0 \leqslant k < n$，若 $K \geqslant 1$，则计算所有顶点 i 在逆树中的深度 depth(i)，需 $O(\log\lceil n/K\rceil)$ 时间、$O(nK)$ 处理器；否则，当 $0<K<1$ 时，则需 $O(\lceil 1/K\rceil \cdot \log n)$ 时间、$O(nK)$ 处理器。

证明　事实上，这个引理是引理 5.2 的推广。现在假定有 nK 个处理器。那么，第五章的算法 5.4 的计算时间为：当 $K \geqslant 1$ 时，第 (1) 步需 $O(1)$ 时间、$O(n)$ 处理器；否则，需 $O(1/K)$ 时间、$O(nK)$ 处理器；第 (2) 步的时间 $T(n)$ 为

$$
\begin{aligned}
T(n) &= \sum_{t=0}^{\log(n-1)-1} (\lceil 2^t/K\rceil) \\
&= \log K + \sum_{t=\log K}^{\log(n-1)-1} (\lceil 2^t/K\rceil) \\
&< \log K + \frac{1}{K}\sum_{t=\log K}^{\log(n-1)-1} 2^t + \log(n-1) - \log K \\
&= O(n/K+\log n)
\end{aligned}
$$

一旦 $F^k(i)$，$i \in V$，$0 \leqslant k < n$ 都计算出来后，depth(i) 的计算如下：对序列 $F^0(i)$，$F^1(i)$，…，$F^{(n-1)}(i)$ 执行二分法搜索，用 $F^{(n-1)}(i)=r_j$ 作为关键字，找出 i 所在树 T_j 的根 r_j，则 r_j 在序列中最左出现的位置 l 就是 i 的深度，即 depth$(i)=l$。当 $0<K<1$ 时，完成二分法搜索需 $O(\lceil 1/K\rceil \cdot \log n)$ 时间。若 $K \geqslant 1$，则采用分组技术，

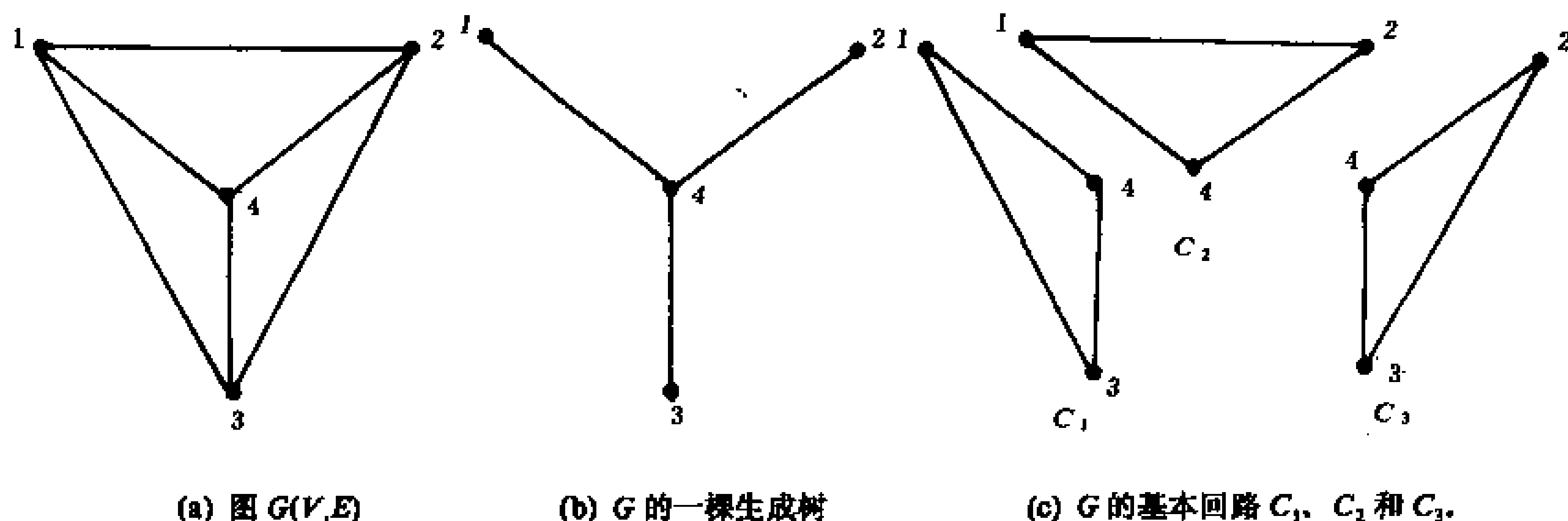
将每个顶点 i 的序列 $F^0(i)$，$F^1(i)$，…，$F^{(n-1)}(i)$ 分成 K 组，每组至多含有序列中的 $\lceil n/K \rceil$ 元素，每组分配一个处理器去执行这组元素的二分法搜索．执行完成后，将第 s 组搜索的结果与第 $s-1$ 组及第 $s+1$ 组的结果相比较 $(2 \leqslant s < K)$，其中必有一个处理器 s 的值同处理器 $s+1$ 的值一致，而同处理器 $s-1$ 的值不一致。当 $s=1$ 时，则仅同第二个处理器的值一致，而当 $s=K$ 时，则仅和第 $K-1$ 个处理器的值不一致。这样，就确定了 r_j 在序列中的最早出现位置 $(r_j = F^{(n-1)}(i))$．因此计算 depth(i) 需 $O(\log \lceil n/K \rceil)$ 时间，$i \in V$．

引理 8.2 在 SIMD－CREW PRAM 上，如果令 $T(V, E')$ 是连通无向图 $G(V, E)$ 的一棵生成树，且以逆树形式存贮．那么在树 T 上找出 G 的 Q 对顶点的最低公共祖先 LCA，需 $O(\lceil Q/nK \rceil \cdot \log n + n/K)$ 时间、$O(nK)$ 处理器 $(1 \leqslant Q \leqslant n^2)$。

证明 由引理 8.1 可知，计算 F^+ 需 $O(n/K + \log n)$ 时间、$O(nK)$ 处理器．找任意一对顶点 (i,j)，$i \in V$，$j \in V$ 的最低公共祖先，可使用两个处理器分别对 F^+ 第 i 行和第 j 行分别执行二分法搜索。直到两个处理器搜索到这两行中第一个相同的元素为止。这个相同的元素 w 就是顶点 i 和 j 的最低公共祖先，即 $w = \mathrm{LCA}(i,j)$．故计算 Q 对顶点的最低公共祖先需 $O(\lceil Q/nK \rceil \cdot \log n + n/K)$ 时间、$O(nK)$ 处理器。其实这个引理是引理 5.3 的一个推广．

8.2 找图的基本回路算法

不失一般性，我们假定图 $G(V, E)$ 是一个无向连通图．令 $T(V, E')$ 是 G 的一棵生成树。对任意一条边 $e = (u,v) \in E - E'$，由边 (u,v)、树 T 上的路径 $[u - \mathrm{LCA}(u,v)]$ 和 $[\mathrm{LCA}(u,v) - v]$ 组成一条封闭的回路。这条回路称为 G 的一条基本回路(Fundamental Cycle)。例如，在图 8.1 中，对于无向连通图 $G(V, E)$，$|V| = 4$(图 8.1(a))，它的一棵生成树是 $T(V, E')$，而且 $E - E' = \{(1,2), (1,3), (2,3)\}$(图 8.1($b$))，图 G 的三条不同的基本回路 C_1，C_2 和 C_3 构成基本回路集 $\Omega = \{C_1, C_2, C_3\}$(图 8.1($c$))．



(a) 图 $G(V,E)$　　(b) G 的一棵生成树　　(c) G 的基本回路 C_1、C_2 和 C_3．

图 8.1 图 G 的生成树和基本回路

根据基本回路的定义，每一条非树边加到树 T 上都形成一条回路，这条回路就是 G 的一条基本回路．因为图 G 中有 $m-(n-1)$ 条非树边，所以，G 有 $m-n+1$ 条基本回路．图 8.1 中的 G，$m=6$，$n=4$，故共有三条基本回路．令 G 的基本回路集为 $\Omega=\{C_1, C_2, \cdots, C_{m-n+1}\}$，其中 C_i 是一条基本回路，$1\leqslant i\leqslant m-n+1$．基本回路的一个重要性质是：图 G 的任何一条回路 C 都是由若干基本回路 C_{i_1}，C_{i_2}，…，C_{i_k} 的"环和"构成的，即 $C=C_{i_1}\oplus C_{i_2}\oplus\cdots\oplus C_{i_k}$，$C_{i_j}\in\Omega$，$1\leqslant i_j\leqslant m-n+1$，$1\leqslant j\leqslant k$．下面，我们非形式地描述找图的基本回路算法[3]:

算法8.1 FINDING FUNDAMENTAL CYCLES OF GRAPHS

输入：无向连通图 $G(V,E)$ 的邻接矩阵，$V=\{1,2,\cdots,n\}$；

输出：n 阶矩阵 LCA^+，F^+ 以及向量 $P^+(1:n)$．

begin

(1) 找出 G 的一棵生成树 $T(V,E')$，T 用逆树形式存贮；

(2) 计算矩阵 LCA^+ 的值．其中 $\mathrm{LCA}^+(i,j)$ 是顶点 i 和 j 在 T 上的最低公共祖先，$i\in V$，$j\in V$；

(3) 对任意一条边 $e\in E-E'$，计算边 e 所在的基本回路

end．

在 SIMD－CREW PRAM 上，算法 8.1 中的第 (1) 步，可以应用第五章求最优 MST 算法构造树 $T(V,E')$；第 (2) 步利用 F^+ 可计算出矩阵 LCA^+ 的值；第 (3) 步实现如下：首先创建一个含有 n 个元素的一维向量 P^+，其中 $P^+(v)=n-1-\mathrm{depth}(v)$，它是顶点 v 在 F^+ 中列的序号，$v\in V$．因此，对于每一条边 $e=(u,v)\in E-E'$，它所在的基本回路 C 是由下列几条路径组成的：

1) 从 F^+ 第 u 行的第 $P^+(u)$ 列到第 $P^+(\mathrm{LCA}(u,v))$ 列上的顶点组成的一条路径 L_1；

2) 从 F^+ 第 v 行的第 $P^+(v)$ 列到第 $P^+(\mathrm{LCA}^+(u,v))$ 列上的顶点构成的一条路径 L_2；

3) 边 (u,v) 组成的一条路径 L_3．

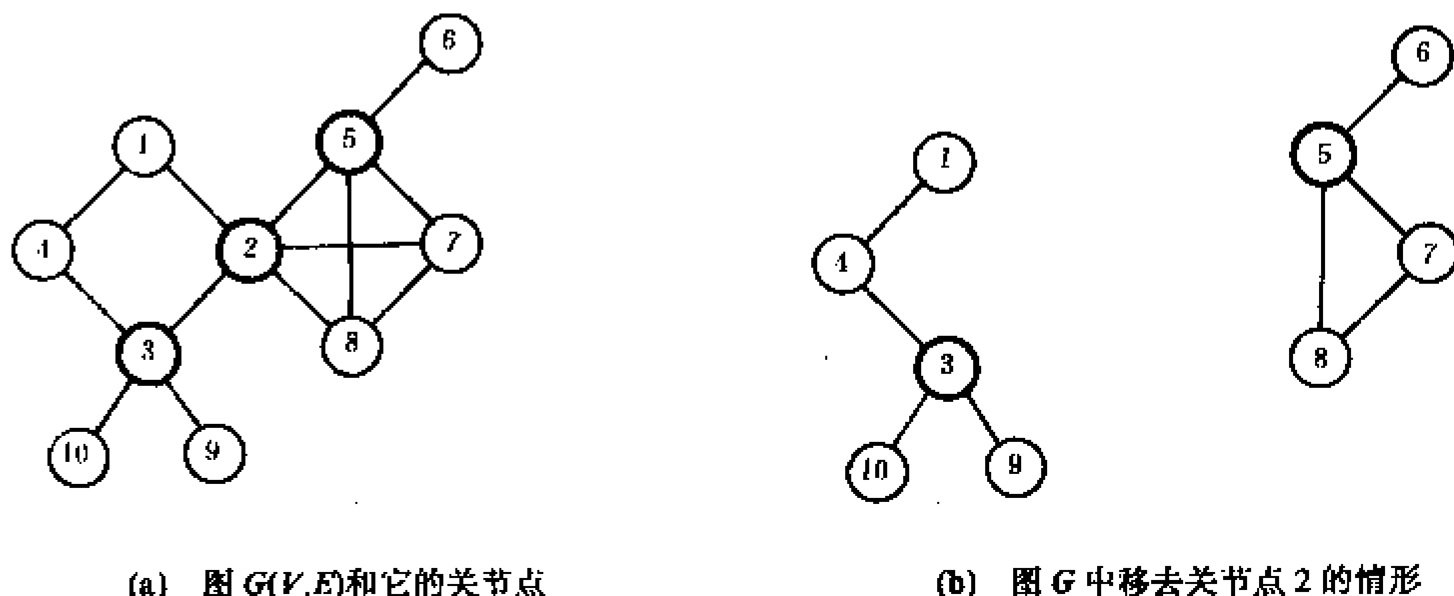
即是说，$C=\bigcup_{i=1}^{3}L_i$．

定理 8.1 在 SIMD－CREW PRAM 上，计算一个无向连通图 $G(V,E)$，$|V|=n$ 的所有基本回路，算法 8.1 需 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器。

证明 由定理 5.6 可知，算法 8.1 的第 (1) 步需 $O(\log^2 n)$ 时间、$O(n^2/\log^2 n)$ 处理器；又由引理 8.2，令 $K=n/\log n$，则第 (2) 步需 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器；第 (3) 步的复杂性由引理 8.1 可知，这一步需 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器。因此整个算法需 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器。

8.3 找图的双连通分支算法

设 $G(V,E)$是一个无向连通图，顶点 $v\in V$，若在 G 中存在两个和 v 不同的顶点 i 与 j，且从 i 到 j 的每一个条路径都经过顶点 v，则 v 称为 G 的**关节点**(Articulation Point)。



(a) 图 $G(V,E)$和它的关节点　　(b) 图 G 中移去关节点 2 的情形

图 8.2 无向连通图及其关节点

这个定义隐含着：若从连通图 G 中移去一个关节点，则将图 G 至少分裂成两个子图。例如，在无向连通的图 $G(V,E)$，$|V|=10$ 中，共有三个关节点，它们是顶点 2、3 和 5。若移去关节点 2，则得到两个连通子图。如图 8.2 所示。

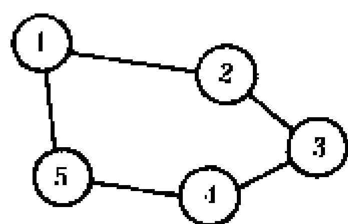


图 8.3 一个双连通图

当且仅当无向连通图 G 不含有关节点时，它是**双连通图**(Biconnected Graph)。例如，图 8.3 所示的图 $G(V,E)$，$|V|=5$，就是一个双连通图，它不存在关节点。

8.3.1 算法的基本原理

在介绍求双连通分支的算法之前，我们先引入一个重要的函数 HLCA(u)，$u\in V$。设 $G(V,E)$是一个无向连通图，$T(V,E')$是 G 的任意一棵逆树（其根为 $r\in V$），对于任何一个顶点 $u\in V$，HLCA(u)＝LCA(u,v)，其中 $e=(u,v)\in(E-E')\cup\{(u,u)\}$，而且 v 必须满足 depth(LCA(u,v))≤depth(LCA(u,v'))，对所有其它的边 $e'=(u,v')\in(E-E')\cup\{(u,u)\}$。

图 8.4 给出 HLCA(u)的一个实例说明。图中的实线及圆圈分别表示逆树的边和顶点，虚线表示与顶点 $u\in V$ 关联的非树边。

为了计算 HLCA(u)，$u\in V$，Tsin 等人[3]利用逆树 T 的顶点先序(Preorder)标号，给出了计算 HLCA(u)的算法，现在就介绍这个算法。

令 $T(V,E')$是 G 的一棵逆树，对树 T 上的每一个顶点 v，指定它的先序标号为 pre(v)。在单机上，深度优先搜索是求先序标号的一个有效算法，但在第三章，我们

曾提及，因为深度优先搜索本质上是顺序的，所以不便于并行化。因此我们采用如下的方法。

为了叙述方便，在树 T 上我们定义关系：$u \preccurlyeq v$ 当且仅当 u 是 v 的一个祖先；$u \prec v$ 当且仅当 u 是 v 的一个真正祖先，$u\in V$，$v\in V$。

引理 8.3 在一棵逆树 $T(V,E')$中，对任意两个顶点 $u\in V$，$v\in V$，$v \preccurlyeq u$ 当且仅当

$\text{pre}(v) \leqslant \text{pre}(u) < \text{pre}(v)+\text{nd}(v)$

这里 nd(v)是 v 的子孙个数。

证明 由先序遍定义，容易证明此引理。

引理 8.4 设无向连通图 $G(V,E)$的一棵逆树是 $T(V,E')$，令$(u,v)\in E-E'$，$(u,w)\in E-E'$，则有：

(1) 若 $\text{pre}(v)<\text{pre}(w)<\text{pre}(u)$，

则 $\text{depth}(\text{LCA}(u,v)) \leqslant \text{depth}(\text{LCA}(u,w))$；

(2) 若 $\text{pre}(v)>\text{pre}(w)>\text{pre}(u)$，

则 $\text{depth}(\text{LCA}(u,v)) \leqslant \text{depth}(\text{LCA}(u,w))$.

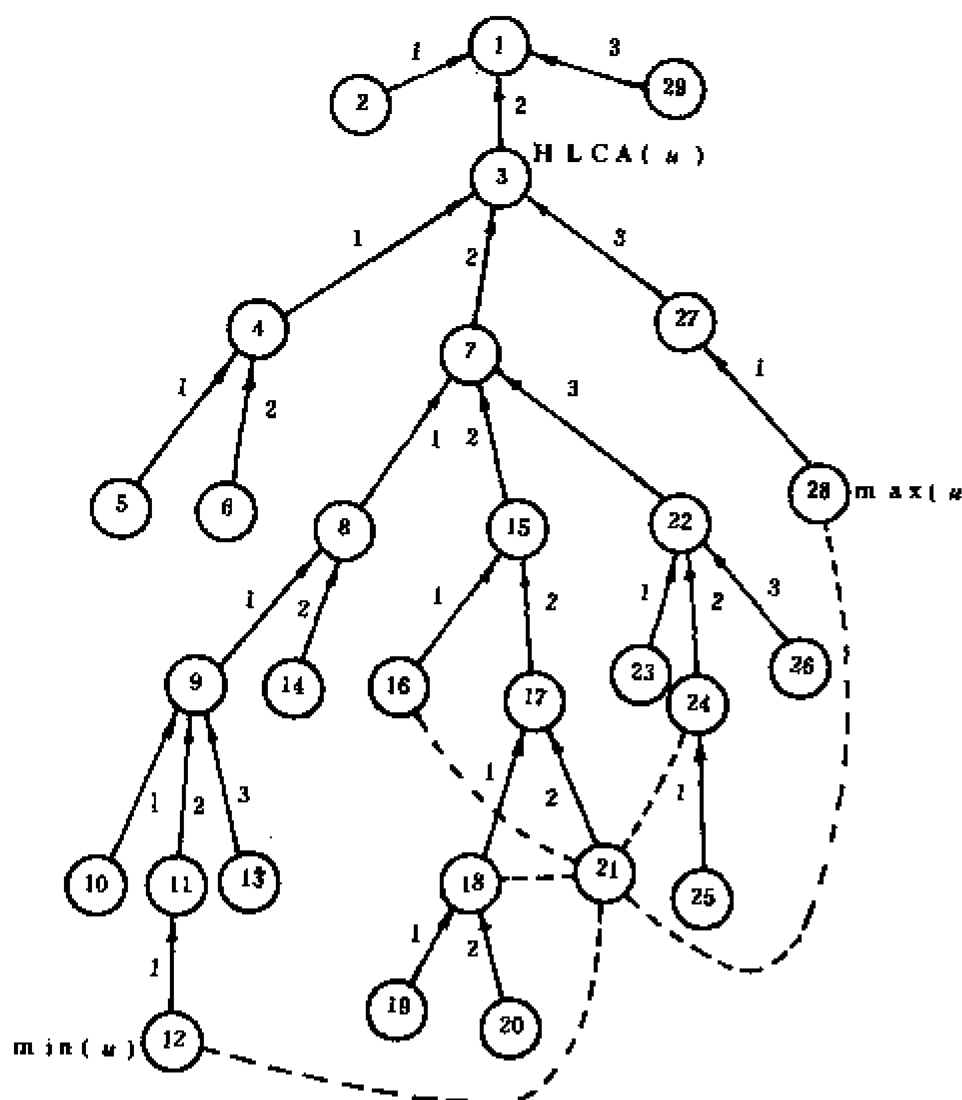


图 8.4 计算 HLCA(u)过程

证明 (1) 根据引理 8.3，$\text{pre}(\text{LCA}(u,v)) \leqslant \text{pre}(v)$，且 $\text{pre}(u) < \text{pre}(\text{LCA}(u,v)) + \text{nd}(\text{LCA}(u,v))$，因此 $\text{pre}(\text{LCA}(u,v)) < \text{pre}(w) < \text{pre}(\text{LCA}(u,v)) + \text{nd}(\text{LCA}(u,v))$。由引理 8.3 可知，$\text{LCA}(u,v) \preccurlyeq w$，所以 $\text{depth}(\text{LCA}(u,v)) \leqslant \text{depth}(\text{LCA}(u,w))$。(2) 可类似地证明。

下面的一个引理即将指出：计算任意一个顶点 $v\in V$ 的 HLCA(u) 问题，可归结到计算两个特定顶点的最低公共祖先问题。

令 $u\in V$，$W=\{v \mid (u,v)\in E-E'\}\cup\{u\}$；

$p\max(u)=v$，这里 $v\in W$ 且 $\text{pre}(v)\geqslant \text{pre}(w)$，$\forall\, w\in W$；

$p\min(u)=v$，这里 $v\in W$ 且 $\text{pre}(v)\leqslant \text{pre}(w)$，$\forall\, w\in W$.

推论 8.1 设无向连通图 $G(V,E)$ 的一棵逆树是 $T(V,E')$，$u\in V$，则

$$\text{HLCA}(u)=\begin{cases}\text{LCA}(u,p\min(u)), & \text{当}\,\text{LCA}(u,p\min(u))\preccurlyeq \text{LCA}(u,p\max(u))\\ \text{LCA}(u,p\max(u)), & \text{当}\,\text{LCA}(u,p\max(u))\preccurlyeq \text{LCA}(u,p\min(u))\end{cases}$$

证明 由引理8.4直接推得。

引理 8.5 设一个无向连通图 $G(V,E)$ 的一棵逆树是 $T(V,E')$，$u\in V$，则

$$\text{HLCA}(u)=\text{LCA}(p\min(u),p\max(u)).$$

证明 由推论8.1可知

$\text{HLCA}(u) \prec p\min(u)$, $\quad\quad$ $\text{HLCA}(u) \prec p\max(u)$

因此

$\text{HLCA}(u) \prec \text{LCA}(p\min(u), p\max(u))$

根据定义

$\text{pre}(p\min(u) \prec \text{pre}(u) \prec \text{pre}(p\max(u))$

这意味着

$\text{pre}(\text{LCA}(p\min(u), p\max(u)) \leqslant \text{pre}(u) < \text{pre}(\text{LCA}(p\min(u), p\max(u)) + \text{nd}(\text{LCA}(p\min(u), p\max(u)))$.

根据引理8.3，

$\text{LCA}(p\min(u), p\max(u)) \prec u$

因而

$\text{LCA}(p\min(u), p\max(u)) \prec \text{LCA}(u, p\min(u))$

且

$\text{LCA}(p\min(u), p\max(u)) \prec \text{LCA}(u, p\max(u))$

由推论8.1，

$\text{LCA}(p\min(u), p\max(u)) \prec \text{HLCA}(u)$

引理 8.6 设 $T(V, E')$ 是一棵逆树，它的顶点已经标上先序标号，则计算 HLCA(u)，$u \in V$ 需 $O(n / K + \log n)$ 时间、$O(nK)$ 处理器。

证明 根据引理 4.4，计算 pmin(u) 及 pmax(u) 需 $O(n / K + \log K)$ 时间、$O(nK)$ 处理器；为了计算 HLCA(u)，需找出 pmax(u)，pmin(u) 的最低公共祖先，一共需要找 n 对这种顶点的最低公共祖先。由引理 8.2，完成这一工作需 $O(n / K + \lfloor 1 / K \rfloor \cdot \log n)$ 时间、$O(nK)$ 处理器。故计算 HLCA(u)，$u \in V$ 需 $O(n / K + \log n)$ 时间、$O(nK)$ 处理器 ($K \geqslant 1$)。

引理 8.7 设 $T(V, E')$ 是一棵有序树，树中的儿子顶点按字典顺序排列，对 V 中任意一个顶点 v，则它的先序标号为：

$$\text{pre}(v) = \sum_{u \in \text{ANC}(v)} \sum_{w \in \text{EBRO}(u)} \text{nd}(w) + \text{na}(v)$$

$$= \sum_{u \in \text{ANC}(v) - \{r\}} \text{nds}(F(u), \text{rank}(u) - 1) + 1 + \text{depth}(v)$$

其中：r 是 T 的根；ANC(v) 是 v 的祖先集合；EBRO(u) 是 u 的兄长集合（兄弟关系是由小到大排）；nd(w) 是 w 的子孙个数；na(v) 是 v 的祖先个数；nds(v,j) 是 v 的前 j 个儿子的子孙总数；rank(v) 是在兄弟中 v 所排位置的序号。

证明 由先序定义直接得到。

实际上，引理8.7给出了计算树T的先序标号算法，现在给出这个算法。

算法8.2 PREORDER TRAVERSAL OF INVERTED TREE

输入：图$G(V, E)$的一棵逆树$T(V, E')$。用二维数组$T(1:2, 1:n)$存贮；

输出：计算结果HLCA(v)，　$v \in \{1, 2, \cdots, n\}$.

begin

(1) 计算T的F^+矩阵和depth向量；

(2) 计算rank(v)； / * 对T的每个顶点$v \in V$的儿子定序 * /

(3) 计算nds(v,j)，$v \in V$，$1 \leqslant j \leqslant n(v)$，其中$n(v)$是$v$的儿子个数；

(4) 计算HLCA(v)，$v \in V$

end .

定理 8.2　在 SIMD－CREW　PRAM 上，计算一棵逆树$T(V, E')$，$|V| = n$的先序标号，算法 8.2 需$O(n / K + \log n)$时间、$O(nK)$处理器。

证明　假定逆树T用一个二维数组$T(1:2, 1:n)$存贮。$V = \{1, 2, \cdots, n\}$，$E' = \{(T(1, i), T(2, i)) \mid 1 \leqslant i \leqslant n\}$，对于根$r$，令$T(2, r) = 0$，在算法 8.2 中，根据引理 8.1，第 (1) 步需$O(n / K + \log n)$时间、$O(nK)$处理器 ($K \geqslant 1$)；第 (2) 步是对有序对$\{(T(2, i), T(1, i)) \mid 1 \leqslant i \leqslant n\}$进行排序，它需要$O(\log n)$时间、$O(n)$处理器[7]．设排序后的$T$矩阵存入$T'(1:2, 1:n)$中，我们将$T'$分成数段．各段的划分规则是：每段的第一行元素都应该是同一个顶点v。这样，每段的第二行顶点就是顶点v的儿子．在顶点i所属的段中，i在第二行所处的位置即是 rank(i)；算法的第 (3) 步计算如下：首先对每个顶点$v \in V$，搜索F^+矩阵的第$(n-1) - \text{depth}(v)$列，计算出v在本列中出现的次数，即得 nd(v)。根据引理 4.4，这一小步需$O(n / K + \log K)$时间、$O(nK)$处理器；然后，用下述公式计算 nds(v,j)，$v \in V$，$1 \leqslant j \leqslant n(v)$。

$$\text{nds}(v, j) = \sum_{\substack{1 \leqslant i \leqslant j \\ s_i \in V \text{且} F(s_i) = v}} \text{nd}(s_i), \quad 1 \leqslant j \leqslant n(v)$$

因为对n个元素计算其部分和$\sum_{1 \leqslant i \leqslant j} a_i$，$(1 \leqslant j \leqslant n)$需$O(\log n)$时间、$O(n)$处理器，但树$T$的每个顶点$v$有$n(v)$个儿子，所以完成上式需$O(\log n(v))$时间、$O(n(v))$处理器。因此，对所有$v \in V$完成上式的计算，需$\max\{\log n(v) \mid v \in V\} = O(\log n)$时间、$O(\sum_{v \in V} n(v)) = O(n)$处理器。这样算法的第 (3) 步需$O(n / K + \log n + \log K)$时间、$O(nK)$处理器 ($K \geqslant 1$)；第 (4) 步的计算使用了引理 8.7 中的公式，开始时假定对任一顶点$v \in V$，nds(v,0) = 0，注意到F^+的第v行的第$(n-1) - \text{depth}(v)$列到第$(n-1)$列的元素都是集合 ANC(v) 的元素，且 nd(v) = depth(v) + 1，根据引理 4.4，这一步需$O(n / K + \log K)$时间、$O(nK)$处理器 ($K \geqslant 1$)。故整个算法需$O(n / K + \log n + \log K)$时间、$O(nK)$处理器 ($K \geqslant 1$)。

定理 8.3　在 SIMD－CREW　PRAM 上，设$T(V,E')$是一个无向连通图$G(V,E)$，$|V| = n$的一棵逆树，则计算所有 HLCA(u)，$u \in V$需$O(n / K + \log n)$时间、$O(nK)$处理器 ($K \geqslant 1$)

证明　由定理 8.2 和引理 8.7 直接推得。

有了前面的一些基本概念后，现在我们来介绍 Tsin 等人基于 SIMD－CREW PRAM 上的最优双连通分支算法。

对一个无向连通图 $G(V, E)$ 来讲，G 的一个双连通分支完全可由它的顶点集确定。因而寻找双连通分支的任务就变成了找双连通分支顶点集的任务。为此，我们在图 $G(V, E)$ 的边之间定义一种关系 **R**。令 $T(V, E')$ 是 G 的一棵逆树，$e_1 = (v, F(v))$，$e_2 = (u, F(u))$，$e_1 \in E'$，$e_2 \in E'$。e_1 和 e_2 具有关系 **R**(记作 $e_1\mathbf{R}e_2$) 当且仅当满足下列条件之一:

(1)　e_2 在树 T 中的 v 到 HLCA(v) 的路径上，或者 e_1 在树 T 中 u 到 HLCA(u) 的路径上;

(2)　$(u,v) \in E - E'$，且在树 T 中 $v \lessdot u$ 或 $u \lessdot v$ 均不成立。

有了这个关系 **R** 的定义，下面我们给出找图的双连通分支算法。

8.3.2　算法的非形式化描述

算法8.3　BICONNECTIVITY ALGORITHM

输入：无向连通图$G(V, E)$，$|V| = n$的邻接矩阵;

输出：图G的所有双连通分支。注意：仅输出每个双连通分支的顶点集。

begin

(1) 找出$G(V,E)$的一棵逆树$T(V, E')$;

(2) 对G的任意一个顶点$v \in V$，计算它的HLCA(v);

(3) 构造一个辅图$G''(E', E'')$，它是一个无向图。其中$(e_1, e_2) \in E''$当且仅当$e_1\mathbf{R}e_2$，$e_1 \in E'$，$e_2 \in E'$;

(4) 找出G''的所有连通分支$\{B_j\}$

end .

引理 8.8　设 $G(V,E)$ 是一个无向连通图，则: (1) 对 G 的每一条边 $e \in E$，存在唯一的一个包含 e 的双连通分支; (2) G 的同一条回路上的边属于同一个双连通分支。

证明　根据关系 **R** 的定义，若 $e_1\mathbf{R}e_2$，则 e_1 和 e_2 同属一条基本回路。事实上，不难证明: **R** 是一个等价关系。因而若 $e_1\mathbf{R}e_2$，$e_2\mathbf{R}e_3$，则 $e_1\mathbf{R}e_3$。也就是说，e_1 和 e_3 也属于同一条回路。

引理 8.9　设 $G(V,E)$ 是一个无向连通图，若 $e_1\mathbf{R}e_2, e_2\mathbf{R}e_3, \cdots, e_{l-1}\mathbf{R}e_l$，则在 G 中存在一条包含 e_1 及 e_l 的回路，$e_i \in E$，$1 \leqslant i \leqslant l$。

证明　由引理8.8直接推得。

定理 8.4　设 $G(V, E)$ 是一个无向连通图，$T(V, E')$ 是 G 的一棵生成树，图 $G''(E', E'')$ 构造为:

$$E'' = \{(e_1, e_2) \mid e_1\mathbf{R}e_2,\ e_1 \in E',\ e_2 \in E'\}$$

在 G'' 中 e 与 e' 属于同一个连通分支，当且仅当在 G 中 e 与 e' 属于同一双连通分支，$e \in E'$，$e' \in E'$。

证明　若 e 及 e' 在 G'' 的同一个连通分支内，则在 G'' 中有一条从 e 到 e' 的路径 e，e_1，e_2,…,e_l，e'。即 $e\mathbf{R}e_1$，$e_1\mathbf{R}e_2$，…，$e_l\mathbf{R}e'$。由引理 8.9，e 及 e' 在 G 的同一回路上，又由引理 8.8 得 e 及 e' 属于 G 的同一个双连通分支。反之，若 e 及 e' 属于 G 的同一双连通分支，则 G 中存在一条包含 e 及 e' 的简单回路[①] C。假定 G 的基本回路集为 Ω，那么存在 Ω 的一个子集 $\{C_i | 1 \leqslant i \leqslant l, C_i \in \Omega\}$，使得 $e \in C_1$，$e' \in C_l$，且 e_i 是 C_i 和 C_{i+1} 的公共边 $(1 \leqslant i < l)$。令 $e_i = (u_i, v_i)$ 是 C_i 中的一条非树边，即 $(u_i, v_i) \in E - E'$。令 $e(u_i)$ 和 $e(v_i)$ 是树的边，即 $e(u_i) = (u_i, F(u_i))$，$e(v_i) = (v_i, F(v_i))$，那么在每一条 C_i 中，我们有：

(1) $e(u_i)\mathbf{R}\,e(v_i)$ 且有 $e_{i-1}\mathbf{R}\,e(u_i)$ 或 $e_{i-1}\mathbf{R}\,e(v_i)$，$e_i\mathbf{R}\,e(u_i)$ 或 $e_i\mathbf{R}\,e(v_i)$；或者

(2) $e_{i-1}\mathbf{R}\,e(u_i)$ 且 $e_i\mathbf{R}\,e(u_i)$;或者

(3) $e_{i-1}\mathbf{R}\,e(v_i)$ 且 $e_i\mathbf{R}\,e(v_i)$。

在上述任何一种情形下，在 G'' 中都存在一条从 e_{i-1} 到 e_i 的路径。尤其是在 G'' 中存在一条从 e 到 e_1 的路径和一条从 e_{l-1} 到 e' 的路径。将所有这些路径连接起来，则得出结论，e 及 e' 同属 G'' 的一个连通分支。

定理 8.5　在 SIMD－CREW PRAM 上，求一个无向连通图 $G(V,E)$ 的双连通分支，算法 8.3 需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器 $(k \geqslant 1)$。

证明　由定理 4.5 及推论 5.1 可知，算法 8.3 的第 (1) 步需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器；第 (2) 步由定理 8.3 可知，需 $(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器；第 (3) 步实现如下：首先构造 G'' 的邻接矩阵 A''，对任意两条边 $e_1 \in E'$ 和 $e_2 \in E'$，且 $e_1 = (u,F(u))$，$e_2 = (v,F(v))$。$A''(e_1, e_2) \leftarrow 1$ 和 $A''(e_2, e_1) \leftarrow 1$ 当且仅当 e_2 在树 T 的 u 到 HLCA(u) 的路径上，或 $(u,v) \in E - E'$ 且 $u \lessdot v$ 和 $v \lessdot u$ 均不成立。又因 $|E'| = n-1$，而矩阵 F^+ 此时又可以利用，所以构造 G'' 需 $O(n/K)$ 时间、$O(nK)$ 处理器；第 (4) 步由定理 4.5 可知，完成这一步需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器。所以，计算图 G 的双连通分支需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器，尤其是当 $K = O(n/\log^2 n)$ 时，算法需$O(\log^2 n)$时间、$O(n^2/\log^2 n)$处理器。这是一个最优算法。

事实上，算法 8.3 中第 (4) 步计算出来的是 G'' 的顶点集划分 $\{B_i\}$。每个 B_j 由 G 的边组成。将这些边的顶点放入一个集合中，则得到的集合就是 G 的一个双连通分支的顶点集。

---

① 在回路 C 中，除首、尾顶点重合外，其它顶点在回路上仅出现一次，则称 C 为简单回路。

8.4 用欧拉遍历技术求双连通分支算法

8.4.1 算法的基本原理

Tarjan 等人[5,6] 在分析了 Tsin 算法的基础上，给出了求图的双连通分支的另一个算法。算法的基本思想亦是将计算图的双连通分支问题归结为计算辅图的连通分支问题。其显著特点是：在树 T 上给出了一种称之为**欧拉遍历** (Euler Tour) 技术，使得树上许多函数的计算（比如每个顶点的先序标号、后序标号及其子孙个数等）都可在 $O(\log n)$ 时间内完成，且使用的处理器数仅为 $O(n)$。所谓树的欧拉遍历技术是指：**对一棵树 $T(V,E')$ 的每条边赋予两条方向相反的并行边，从而得到一个有向欧拉图，然后计算树的一些函数。**即将介绍的 Tarjan 算法，和算法 8.3 相比较、至少有两点改进：一是构造的辅图 G'' 含有的边数较少，辅图的计算较算法 8.3 容易；更重要的是，由于在树上引入欧拉遍历技术，使得树上一些基本函数的计算只需 $O(\log n)$ 时间、$O(n)$ 处理器。

8.4.2 算法的非形式化描述

设 $G(V,E)$，$|V|=n$ 是一个无向连通图。若 $\mathbf{R}$ 是 G 中边之间的一种关系，$\mathbf{R}$ 定义为 $e_1\mathbf{R}e_2$ 当且仅当 $e_1=e_2$ 或者 e_1 和 e_2 都在 G 的一条简单回路上，$e_1\in E$，$e_2\in E$。显然边集合 E 上的关系 $\mathbf{R}$ 是一个等价关系。由 $\mathbf{R}$ 导出的等价类是 G 的一个双连通分支。

和前面一节类似，我们定义 G 的一个辅图 G'，G' 的顶点由 G 的边组或，G' 的一个连通分支对应 G 的一个双连通分支。令 $S\subseteq E$ 是 G 的一个边集合，则由 S 导出 G 的一个子图是 G 的一个双连通分支，当且仅当 G' 的顶点集 S 导出的子图是 G' 的一个连通分支。令 $T(V,E')$ 是 $G(V,E)$ 的一棵生成树，我们用 $v\rightarrow w$ 表示一条树边，即 v 是 w 的父亲 $F(w)$。则 G' 的边是由下列的边组成的：

1) $((u,w),(v,w))$，这里 $u\rightarrow w$ 是T 的边，$(v,w)\in E-E'$，且 $\mathrm{pre}(v)<\mathrm{pre}(w)$；
2) $((u,v),(x,w))$，这里 $u\rightarrow v$ 和 $x\rightarrow w$ 是 T 的边，$(v,w)\in E-E'$ 且在 T 中 $v\lessdot w$ 和 $w\lessdot v$ 均不成立；
3) $((u,v),(v,w))$，这里 $u\rightarrow v$ 和 $v\rightarrow w$ 是 T 的边，G中存在某条边将 w 的一个子孙与一个不是 v 的子孙的顶点联接起来。

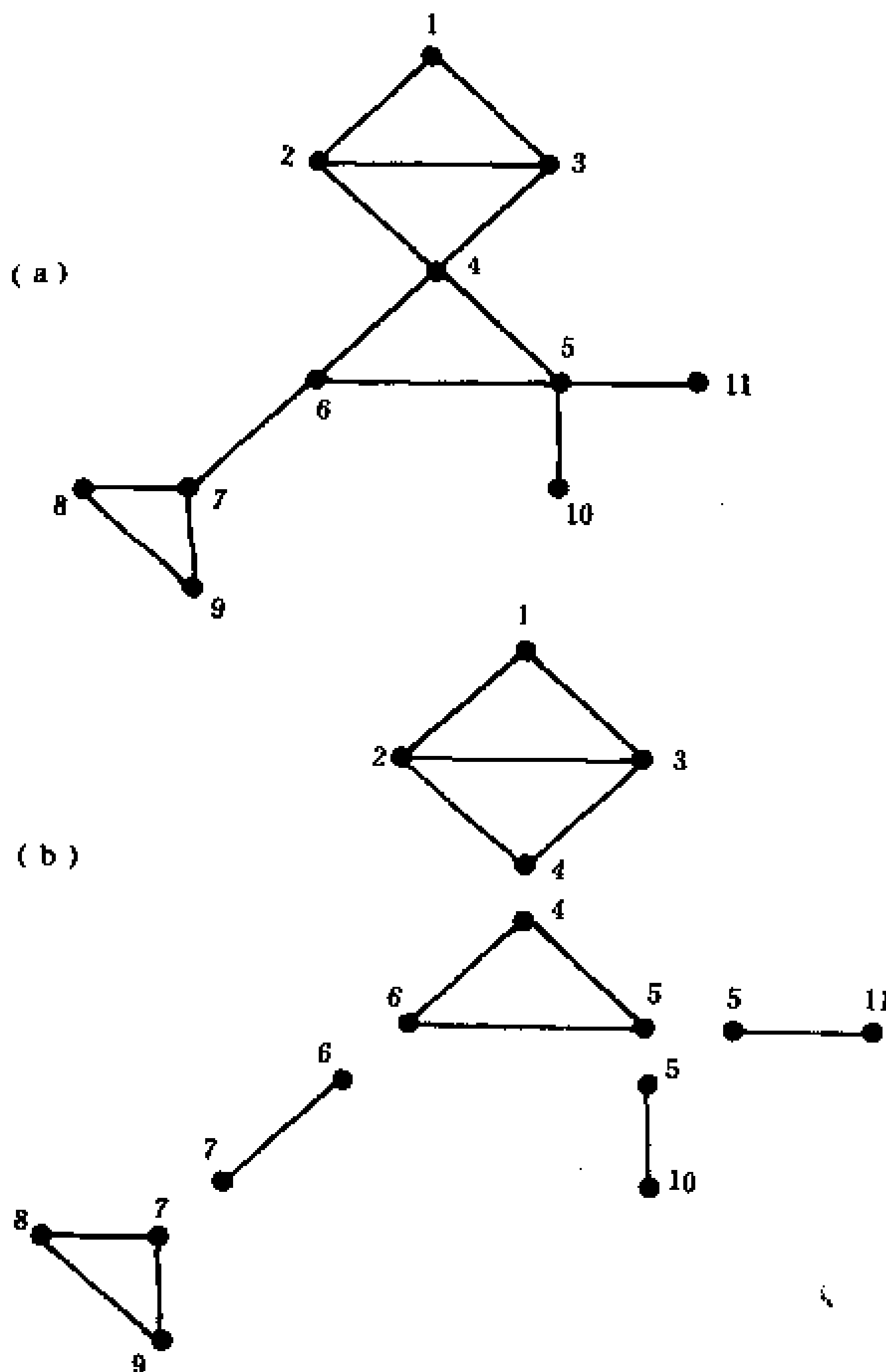
构造 G' 的基本原则是：对 G 的一棵生成树 T，每条非树边 $e\in E-E'$ 确定了 G 的一条基本回路。同一条基本回路上的边在同一个双连通分支内。我们添加足够多的边到 G' 中，使得对应 G 的一条回路的所有边在 G' 的同一个连通分支内。

在图 8.5 中，给出了一个无向连通图 $G(V,E)$，$|V|=11$ 和它的双连通分支．G 的双连通分支在图 (b) 示出。图 8.6 给出了图 G 的辅图 G' 的构造过程。图 (a) 含有 G 的一棵生成树，图 (b) 是构造的结果 —— 辅图 G'。

定理 8.6 设 $G(V,E)$ 是一个无向连通图．$T(V,E')$ 是 G 的一棵生成树．则 G 的两条边在同一双连通分支内，当且仅当 G' 中对应这两条边的顶点在同一连通分支内．

证明 对任意一条边$(x,y)\in E-E'$，定义 G 的一条简单回路，即(x,y)所在的基本回路。所有的基本回路组成了基本回路集 Ω。图 G 的任意一条回路都是由 Ω 内的一些元素的“环和$\oplus$”构成的。

定义关系 $\mathbf{R}'$为：$e_1\mathbf{R}'e_2$ 当且仅当 e_1 与 e_2 是 G 的一条基本回路上的边。令 $\mathbf{R}'^*$ 是 $\mathbf{R}'$的自反传递闭包。我们可以断言 $\mathbf{R}'^*=\mathbf{R}$。这是因为 $\mathbf{R}$ 是 E 上的一个等价关系，且 $\mathbf{R}'\subseteq\mathbf{R}$，所以 $\mathbf{R}'^*\subseteq\mathbf{R}$。接着再证明 $\mathbf{R}\subseteq\mathbf{R}'^*$。若 $e_1\mathbf{R}e_2$，则 e_1 与 e_2 在 G 的同一条简单回路 C 上。而 C 是由 Ω 的一些元素环和而成，即 $C=C_1\oplus C_2\oplus\cdots\oplus C_k$，$C_i\leqslant\Omega$，$1\leqslant i\leqslant k$。不失一般性，假定 C_1，C_2，…，C_k 具有这样的性质：使得 C_i $(i>1)$至少与某个$C_j(j<i)$有一条公共边。那么，通过对 k 进行归纳，我们很容易证得在 $\mathbf{R}'^*$内 C_1，C_2，…，C_k 的边是等价的，即 e_1 $\mathbf{R}'^*e_2$，故 $\mathbf{R}\subseteq\mathbf{R}'^*$。因此断言 $\mathbf{R}=\mathbf{R}'^*$成立。



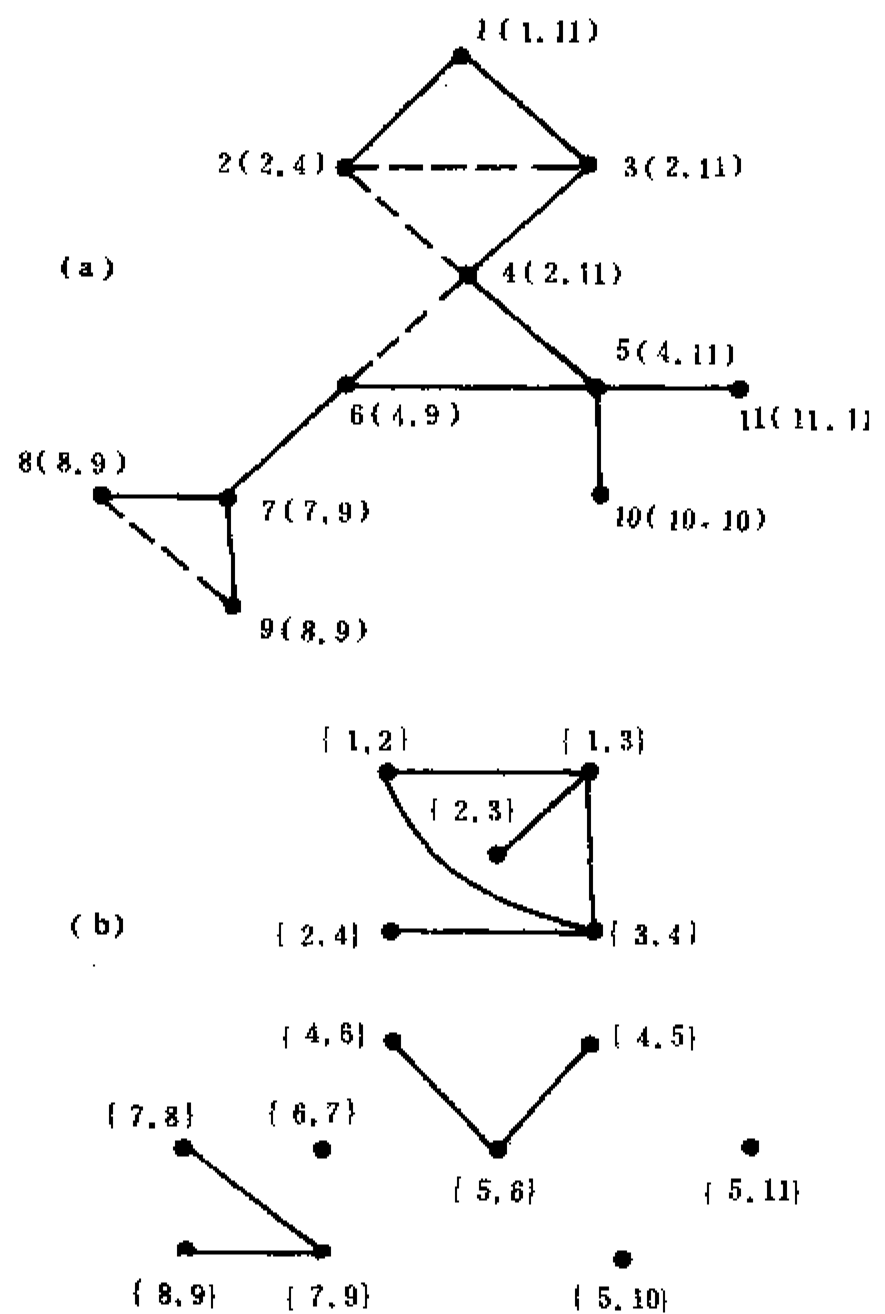
(a) 无向图 G

(b) 图 G 的双连通分支

(顶点 4，5，6 和 7 是关节点)

(边(6,7)，(5,10)和(5,11)是桥)

图 8.5 无向图的双连通分支及关节点



(a) G 的生成树

(虚线为非树边，括号中数分别是 low 和 high 值)

(b) 辅图 G'

图 8.6 G 的辅图 G'的构造过程

令 (u,v) 及 (x,w) 是 G' 的两个邻接顶点，若条件 1) 成立，则 (u,v) 在 (x,w) 定义的基本回路上 $(x = v)$。若条件 2) 成立，则 (u,v) 及 (x,w) 在 (v,w) 定义的基本回路上。若条件 3) 成立，比如说，(y,z) 是这样的一条边，y 是 w 的一个子孙，而 z 不是 $v = x$ 的一个子孙，那么 (u,v) 及 (x,w) 在 (y,z) 定义的基本回路上。因此，在任何条件下，(u,v) 及 (x,w) 均在 G 的同一个双连通分支内。

反之，令 $(x,y) \in E - E'$，它定义了一条含边 (x,y) 的基本回路。令 $z = \mathrm{LCA}(x,y)$。不失一般性，假定 $\mathrm{pre}(x) < \mathrm{pre}(y)$，根据条件 1)，$(x,y)$ 及 $(F(y),y)$ 在 G' 中是邻接顶点。又由条件 3) 可知，边 (x,y) 一定存在。这就意味着：在树上从 x 到 $\mathrm{LCA}(x,y)$ 路径上的任何两条边，对应在 G' 上的顶点都是邻接顶点。类似地，在树上从 y 到 $\mathrm{LCA}(x,y)$ 路径上的任意两条边，对应在 G' 上的顶点都是邻接顶点。若 $z = x$，则在树中从 x 到 z 的路径为空；否则在树上 $x \nless y$ 或 $y \nless x$ 均不成立。又由条件 2) 得 $(F(x),x)$ 及 $(F(y),y)$ 在 G' 中是邻接顶点。因此，同一条基本回路上的边在 G' 的同一个连通分支内。

算法8.4 BICONNECTIVITY ALGORITHM BY EULER TRAVERSAL

输入：图$G(V,E)$的邻接表，且$V = \{1, 2, \cdots, n\}$，无向边$(i, j) \in E$用两条有向边 $<i,j>$ 和 $<j,i>$ 表示，$i,j \in V$；

输出：图G的双连通分支。

begin

(1) 构造$G(V,E)$的一棵生成树$T(V,E')$，计算每个顶点$v \in V$在T中的先序标号$\mathrm{pre}(v)$及其子孙个数$\mathrm{nd}(v)$；

/ * 一个顶点u是另一个顶点v的子孙，当且仅当$\mathrm{pre}(v) \leqslant \mathrm{pre}(u) \leqslant \mathrm{pre}(v) + \mathrm{nd}(v) - 1$，

其中$\mathrm{nd}(v)$可通过对T后序遍历得到，$\mathrm{nd}(v) = 1 + \sum_{\substack{(F(w),w) \in E \\ F(W) = V}} \mathrm{nd}(w)$，（$v$是它自己的

子孙。）* /

(2) 对每个顶点$v \in V$，计算最小先序标号顶点$\mathrm{low}(v)$及最大先序标号顶点$\mathrm{high}(v)$；它们满足：

$\mathrm{low}(v) = \{u \mid \mathbf{min}\{\mathrm{pre}(v), \mathrm{low}(u) \mid v \to u$是$T$的边，$(v,u) \in E - E'\}\}$；

$\mathrm{high}(v) = \{u \mid \mathbf{max}\{\mathrm{pre}(v), \mathrm{low}(u) \mid v \to u$是$T$的边，$(v,u) \in E - E'\}\}$；

/ * 可以通过对T后序遍历得到 * /

(3) 构造G'的一个子图G''，G''的顶点由T的边组成，G''的边由满足条件2)及条件3)的边组成。对每条边$(w,v) \in E - E'$，若$\mathrm{pre}(v) + \mathrm{nd}(v) \leqslant \mathrm{pre}(w)$，则把边$((F(v),v), (F(w),w))$加入到$G'$中；

/ * 按条件2) * /

(4) 找出G''的连通分支；

(5) 将T上边的等价关系扩展到非树边$e \in E - E'$上去，对每一条边 $e = (v,u) \in E - E'$，且$\mathrm{pre}(v) < \mathrm{pre}(u)$，定义$(v,u)$等价于$(F(u),u)$

/ * 按条件1) * /

end.

8.4.3 算法在 SIMD 共享存贮模型上的实现

Tarjan 等人在 SIMD－CRCW PRAM 上，实现了算法 8.4，得到一个 $O(\log m)=O(\log n)$ 时间、$O(m+n)$ 处理器的并行算法。

设算法的输入表示为：$V=\{1,2,\cdots,n\}$，$G(V,E)$ 的每条无向边 $(i,j)\in E$ 用两条方向相反的并行有向边 $<i,j>$ 及 $<j,i>$ 表示。每个顶点 $i\in V$ 有一个射出边的链表，其中 adj(i) 是表头指针，指向 i 的第一条射出边 $<i,j>$，或者当 i 没有射出边时，表头指针为空 (NULL)。若还有从 i 出发的射出边，则 next($<j,i>$) 指向该射出边；否则，置 next($<i,j>$) 为空。此外，每条边 $<i,j>$ 还有一个指针指向边 $<j,i>$。对每个顶点 i 和每条边 $<i,j>$ 都分派一个处理器，分别是 PE(i)，PE(i,j)，$1\leqslant i,j\leqslant n$。

算法 8.4 的第 (1) 步实现首先是构造一棵生成树 $T(V,E')$，这可运用推论 5.2 及 Shiloach 算法[8] 直接得到，需 $O(\log n)$ 时间、$O(m+n)$ 处理器。但这样产生的一棵生成树 T，它还没有根，我们指定一个顶点为树根，然后在这棵根树上计算每个顶点 v 的先序标号 pre(v) 及子孙个数 nd(v)，$v\in V$。为此，我们将树边也用邻接表形式存贮。具体的构造过程如下：

算法 8.5 CONSTRUCTING LIST OF TREE

输入：图 $G(V,E)$ 的邻接表，$V=\{1,2,\cdots,n\}$；

输出：G 的一棵生成树 $T(V,E)$ 的邻接表。

```
procedure Incident__List__Tree ;
  begin
  (1) for each i,j : (i, j)∈E pardo          / * 赋初值 * /
    (1.1) treenext( < i, j > ) ← next( < i, j > ) ;
    (1.2) treenext( < j, i > ) ← next( < j, i > )
      endfor ;
  (2) for k ← 1 to ⌈logm⌉ do
     (2.1) for each i, j: (i, j)∈E pardo
       (2.1.1) if (treenext( < i, j> ) ≠ NULL) and (treenext( < i, j> )∉E′)
                  then treenext( < i, j > ) ← treenext(treenext( < i, j > )
             endif ;      / * 跳过每条树边的下一条非树边的邻接边 * /
       (2.1.2) if (treenext( < j, i > ) ≠ NULL) and (treenext( < j, i > )∉E′)
                then treenext( < j,i > ) ← treenext(treenext ( < j, i > ))
             endif
         endfor
     endfor ;
  (3) for each i: 1 ≤ i ≤ n pardo      / * 构造树的邻接表表头 * /
        if (adj(i) = NULL or (adj(i)∈E′)
            then treeadj(i) ← adj(i)
            else treeadj(i) ← treenext(adj(i))
```

8.4.3 算法在SIMD共享存贮模型上的实现

Tarjan 等人在 SIMD－CRCW PRAM 上，实现了算法 8.4。得到一个 $O(\log m) = O(\log n)$ 时间、$O(m+n)$ 处理器的并行算法。

设算法的输入表示为：$V=\{1,2,\cdots,n\}$，$G(V,E)$ 的每条无向边 $(i,j)\in E$ 用两条方向相反的并行有向边 $<i,j>$ 及 $<j,i>$ 表示。每个顶点 $i\in V$ 有一个射出边的链表，其中 adj(i) 是表头指针，指向 i 的第一条射出边 $<i,j>$，或者当 i 没有射出边时，表头指针为空 (NULL)，若还有从 i 出发的射出边，则 next($<j,i>$) 指向该射出边；否则，置 next($<i,j>$) 为空。此外，每条边 $<i,j>$ 还有一个指针指向边 $<j,i>$。对每个顶点 i 和每条边 $<i,j>$ 都分派一个处理器，分别是 PE(i)，PE(i,j)，$1\leqslant i,j\leqslant n$。

算法 8.4 的第 (1) 步实现首先是构造一棵生成树 $T(V,E')$，这可运用推论 5.2 及 Shiloach 算法[8] 直接得到，需 $O(\log n)$ 时间、$O(m+n)$ 处理器。但这样产生的一棵生成树 T，它还没有根，我们指定一个顶点为树根，然后在这棵根树上计算每个顶点 v 的先序标号 pre(v) 及子孙个数 nd(v)，$v\in V$。为此，我们将树边也用邻接表形式存贮。具体的构造过程如下：

算法 8.5 CONSTRUCTING LIST OF TREE

输入：图$G(V,E)$的邻接表，$V=\{1,2,\cdots,n\}$；

输出：G的一棵生成树$T(V,E)$的邻接表。

```
procedure Incident__List__Tree ;
  begin
  (1) for each i,j : (i, j)∈E pardo          / * 赋初值 * /
   (1.1) treenext( < i, j > ) ← next( < i, j > ) ;
   (1.2) treenext( < j, i > ) ← next( < j, i > )
      endfor ;
  (2) for k ← 1 to ⌈logm⌉ do
    (2.1) for each i, j: (i, j)∈E pardo
      (2.1.1) if (treenext( < i, j> ) ≠ NULL) and (treenext( < i, j> )∉E′)
                   then treenext( < i, j > ) ← treenext(treenext( < i, j > )
             endif;        / * 跳过每条树边的下一条非树边的邻接边 * /
      (2.1.2) if (treenext( < j, i > ) ≠ NULL) and (treenext( < j, i > )∉E′)
               then treenext( < j,i > ) ← treenext(treenext ( < j, i > ))
             endif
          endfor
      endfor ;
  (3) for each i: 1 ≤ i ≤ n pardo        / * 构造树的邻接表表头 * /
          if (adj(i) = NULL or (adj(i)∈E′)
              then treeadj(i) ← adj(i)
              else treeadj(i) ← treenext(adj(i))
```

$$\text{pre}(k_0) = \min \{\text{pre}(\text{local_low}(k')) \mid \text{pre}(j) \leqslant \text{pre}(k') \leqslant \text{pre}(j) + \text{nd}(j) - 1\}$$

则 $\text{low}(j) = k_0$.

为了计算每个 $j \in V$ 的 $\text{low}(j)$，我们定义一个顶点辅助数组 I，开始时 $I(i) \leftarrow i$；然后对每个顶点的先序标号按非降次序排序。在排序过程中，若 $\text{pre}(i)$ 与 $\text{pre}(j)$ 交换位置时，则 $I(i)$ 与 $I(j)$ 也跟着交换位置，在排序结束后，令得到的先序标号数组为 PRE；然后每个处理器 j 在数组 PRE 上执行二分法搜索，以确定 j 在这个数组上的位置 $\text{locate}(j)$。则 $I(\text{locate}(j)) = j$. 所以，若

$$\text{pre}(I(k_0)) = \min \{\text{pre}(\text{local_low}(I(k')) \mid \text{locate}(i) \leqslant k' \leqslant \text{locate}(i) + \text{nd}(i) - 1\}$$

则

$$\text{low}(j) = I(k_0)$$

不失一般性，假定 n 是 2 的幂，又定义一个辅助数组 global_low:

$$\text{global_low}(i,j) = \text{local_low}(I(k_0))$$

其中:

$$\text{pre}(\text{local_low}(I(k_0))) = \min_{k'} \{\text{pre}(\text{local_low}(I(k')) \mid i \leqslant k' \leqslant j\},\ i,j \in V$$

由此可知，顶点 $I(i), I(i+1), \cdots, I(j)$ 中的 local_low 值最小的顶点 $\text{local_low}(I(k_0))$, $i \leqslant k_0 \leqslant j$, 即是 $\text{global_low}(i,j)$。global_low 的计算过程如下:

算法8.6 COMPUTING GLOBAL LOW

```
  输入:  pre(1:n) 和 local_low(1:n);
  输出:  global_low(1:n).
procedure Compute_global_low;
  begin
  (1) for each i: 1 ≤ i ≤ n pardo
         global_low(i, i) ← local_low(I(i))
      endfor;
  (2) for l ← 1 to logn do
         for each k: 0 ≤ k ≤ n/2^l − 1 pardo
            if pre(global_low(k2^l + 1, (2k − 1)2^(l−1))) >
                              pre(global_low((2k − 1)2^(l−1), (k + 1)2^l)
              then global_low(k2^l, (k + 1)2^l) ← global_low((2k − 1)2^(l−1),
                                                  (k + 1)2^l)
              else global_low(k2^l + 1,(k + 1)2^l) ← global_low(k2^l + 1,(2k − 1)2^(l−1))
            endif
         endfor
      endfor
```

end .

在数组 I 上计算出 global_low 后，即可计算 low(j)，$j\in V$。只要注意到 locate($I(j)$) $= j$，则顶点 $I(j+1), I(j+2), \cdots, I(j+\mathrm{nd}(j)-1)$ 都是 j 的儿子顶点。因为对任意一个 k，$\mathrm{pre}(j)\leqslant \mathrm{pre}(I(k))\leqslant \mathrm{pre}(j)+\mathrm{nd}(j)-1$，所以计算 low($j$) 是在 I 中从 locate(j) 到 locate(j) + nd(j) − 1 这一段上进行的。每个 low(j) 的计算过程如下:

算法8.7 COMPUTING LOCAL LOW

输入: locate(1: n)及nd(1: n) ;

输出: low(i) , $1\leqslant i\leqslant n$.

```
procedure low ;
  begin
  (1) for each i : 1 ≤ i ≤ n pardo
            l(i) ← locate(i) ;  h(i) ← locate(i) + nd(i) − 1 ;
            low(i) ← n + 1 ;        / * 任意指定一个大于n的值 * /
            flag(i) ← true        / * flag(i)为假时，表示low(i)已经算出 * /
      endfor ;
  (2) for k ← 1 to logn do
        for each i: (1 ≤ i ≤ n) & flag(i) pardo
      (2.1)  if (l(i) − 1) mod 2^k ≠ 0
                 then if pre(global_low(l(i), l(i) + 2^(k−1) − 1)) < pre(low(i))
                        then low(i) ← global_low(l(i), l(i) + 2^(k−1) − 1);
                             l(i) ← l(i) + 2^(k−1)
                      endif
             endif ;
      (2.2)  if h(i) mod 2^k ≠ 0
                 then if pre(global_low(h(i) − 2^(k−1) + 1, h(i))) < pre(low(i))
                        then low(i) ← global_low(h(i) − 2^(k−1) + 1, h(i))
                      endif
             endif ;
      (2.3)  if l(i) > h(i) then flag(i) ← false endif
        endfor
    endfor
  end .
```

显然，对每个顶点 j，过程 low(j) 的计算时间只需 $O(\log n)$ 且使用的处理器数目为 $O(n)$。而 high(j) 亦是类似的。

算法的第 (3) 步构造辅图 G''，仅需 $O(1)$ 时间、$O(m)$ 处理器．这是因为每条边赋予了一个处理器，每条边的测试仅需 $O(1)$ 时间．

算法的第 (4) 步寻找 G'' 的连通分支，可以采用第四章介绍过的算法，需 $O(\log n)$ 时间、$O(n+m)$ 处理器．这是因为 G'' 的顶点数即是树的边数，所以同一个连通分支的顶点都有相同的 D 值。

算法第 (5) 步将树边上的等价关系推广到非树边集合 $E-E'$ 上．若对每条边 $<i,j>$ 有 $\mathrm{pre}(i)<\mathrm{pre}(j)$，$<i,j>\in E$，则 $D(<i,j>)\leftarrow D<F(j),j>$。这一步需 $O(1)$ 时间、$O(m)$ 处理器．

定理 8.7　在 SIMD－CRCW PRAM 上，计算一个无向连通图 $G(V,E)$，$|V|=n$ 的双连通分支，算法 8.4 需 $O(\log n)$ 时间、$O(n+m)$ 处理器．

证明　在算法 8.4 中，第 (1) 步构造 G 的一棵生成树 T，需 $O(\log n)$ 时间、$O(n+m)$ 处理器，而利用欧拉遍历技术计算树顶点的先序标号、子孙个数，需 $O(\log n)$ 时间、$O(n)$ 处理器；第 (2) 步需 $O(\log n)$ 时间、$O(\max\{n,m\})=O(m)$ 处理器；第 (3) 步需 $O(1)$ 时间、$O(m)$ 处理器；第 (4) 步需 $O(\log n)$ 时间、$O(n+m)$ 处理器；第 (5) 步需 $O(1)$ 时间、$O(m)$ 处理器．故整个算法需 $O(\log n)$ 时间、$O(n+m)$ 处理器．

推论 8.2　在 SIMD－CREW PRAM 上，对一个无向连通图 $G(V,E)$，$|V|=n$，若 G 的输入是邻接矩阵表示，则求 G 的双连通分支算法 8.4，需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器 $(K\geqslant 1)$．

证明　算法 8.4 的第 (1) 步构造生成树 T 及第 (4) 步计算 G'' 的连通分支，由定理 4.5 及推论 5.1 得知，完成这工作需 $O(n/K+\log n)$ 时间、$O(nK)$ 处理器；而算法的第 (3) 步及第 (5) 步可采用分组技术，每个处理器至少有 $O(n^2/nK)=O(n/K)$ 条边，故完成这两步各需 $O(n/K)$ 时间、$O(nK)$ 处理器；在第 (1) 步计算每个树顶点的先序标号、子孙个数和第 (2) 步计算 low 及 high 的值，都要求树用邻接表形式存贮．具体实现如下：首先通过 G 的邻接矩阵构造 G 的邻接表．对 G 的边按字典序进行排序，采用 Cole 算法[7] 需 $O(\log m)$ 时间、$O(m)$ 处理器；然后将排序后的边序列构造成邻接表；再按本节开始叙述的，从 G 的邻接表（又称出边表）中构造出树的邻接表，需 $O(\log m)$ 时间、$O(m)$ 处理器．故第 (1) 步中的计算先序标号和子孙个数及整个第 (2) 步计算，需 $O(\log m)=O(\log n)$ 时间、$O(m+n)$ 处理器。所以，整个算法 8.4 需要 $O(\log^2 n+n/K)$ 时间、$O(nK)$ 处理器．这同定理 8.5 的结论是一致的．

8.5 桥 的 算 法

8.5.1 桥的基本性质

设 $G(V,E)$ 是一个无向连通图，$V=\{1,2,\cdots,n\}$．若从图 G 中删除边 $e\in E$ 之后，G 就不连通，则边 e 称为 G 的一座桥 (Bridge)．例如，在图 8.5 中，图 G 是连通的，若删去边 (6,7)、或边 (5,10)、或边 (5,11)，则留下的图就不再是连通图，故边 (6,7)、(5,10) 和 (5,11) 是图 G 的桥．

令 G 的一棵生成树为 $T(V,E')$，树 T 的边为 $<i,F(i)>\in E'$，$i\in V$。即树 T 以逆树形式存贮，其中根 r 有 $F(r)=r$. 对于任何一条非树边 $e\in E-E'$，把它加入到 T 中，则形成 G 的一个基本回路 C. 由于从 C 中去掉非树边 e 后并不影响图 G 的连通性，故边 e 不可能是 G 的桥，因而只有生成树的边 $<i,F(i)>E'$ 才有可能成为 G 的桥. 那么什么样的树边才是 G 的一座桥呢? 请看下面的定理.

定理 8.8 设 $T(V,E')$ 是一个无向连通图 $G(V,E)$ 的一棵生成树。对任意一条树边 $<i,F(i)>\in E'$, $(i\neq r)$ 是 G 的一座桥，当且仅当这条边是 i 的子孙和非 i 子孙之间的唯一连接边。即对 i 的任何子孙 j，在 G 中不存在这样的边 (j,k)，其中 k 不是 i 的子孙，$i\in V$.

证明 令 $<i,F(i)>$ 是 G 的一座桥. 若存在这样一条边 $(j,k)\in E-E'$，且 j 是 i 的子孙，k 不是 i 的子孙. 那么在树 T 中，$z=\text{LCA}(k,j)$ 是 j 和 k 的最低公共祖先，又因为 k 不是 i 的子孙，所以 z 必然也是 i 的祖先。故在树 T 中边 $<i,F(i)>$ 在 j 到 z 的路径上。若将边 (j,k) 加入到 T 中，则形成 G 的一条基本回路 C'，且边 $<i,F(i)>$ 在 C' 上，从 C' 上去掉边 $<i,F(i)>$ 并不影响 G 的连通性，故 $<i,F(i)>$ 不是桥. 这就与 $<i,F(i)>$ 是桥的定义矛盾.

反之，若边 $e=<i,F(i)>\in E'$ 是一条树边，它是 i 的子孙和非 i 子孙之间的唯一连接边. 令 i 的子孙顶点集合 $U=\{u \mid i\leqslant u$ 在 T 中$\}$，则非 i 子孙顶点集为 $V-U$. 若从 G 中删除边 e 以后，则任意顶点 $x\in U$ 和顶点 $y\in V-U$ 之间不存在路径，即 G 变成不连通了. 根据桥的定义，所以 e 是 G 的一座桥.

根据定理 8.8, Savage 在 SIMD－CREW PRAM 上，给出了一个计算 G 的桥算法[1,2]，此算法需 $O(\log^2 n)$ 时间、$O(n^2\log n)$ 处理器. 这里我们介绍 Tsin 等人[3] 给出的另一个算法。

定理 8.9 设 $T(V,E')$ 是无向连通图 $G(V,E)$ 的一棵生成树，且用逆树形式存贮。对任意一条树边 $<i,F(i)>\in E'$，$i\neq r$，r 是树根顶点，$<i,F(i)>$ 是 G 的一座桥，当且仅当 i 的任意一个子孙 u，在 G 中不存在非树边 $(u,v)\in E-E'$ 且 $\text{depth}(\text{LCA}(u,v))<\text{depth}(i)$。

证明 令树边 $<i,F(i)>, i\neq r$ 是 G 的一座桥. 若存在一条边 $(u,v)\in E-E'$ 且 u 是在 T 中的子孙，而且还有 $\text{depth}(\text{LCA}(u,v))<\text{depth}(i)$，即么将边 (u,v) 加入 T 中后，在树中从 u 到 $\text{LCA}(u,v)$ 的路径、从 $\text{LCA}(u,v)$ 到 v 的路径和边 (u,v) 形成一条包括边 (u,v) 的基本回路. 而边 $<i,F(i)>$ 在 u 到 $\text{LCA}(u,v)$ 路径上. 这同 $<i,F(i)>$ 是 G 的桥相矛盾.

反之，若边 $e\in E$ 不是桥，则它属于某条基本回路. 如果 $e=(u,v)\notin E-E'$，那么在树 T 中，边 e 或在 v 到 $\text{LCA}(u,v)$ 路径上，或在 $\text{LCA}(u,v)$ 到 u 的路径上。这就意味着:

$$\text{depth}(u)\geqslant\text{depth}(i)>\text{depth}(F(i))\geqslant\text{depth}(\text{LCA}(u,v))$$

或

$$\text{depth}(v)\geqslant\text{depth}(i)>\text{depth}(F(i))\geqslant\text{depth}(\text{LCA}(u,v))$$

在任何一种情况下，都至少存在一条边 $(u,v)\in E-E'$ 且 u 是 i 的子孙，$\text{depth}(\text{LCA}(u,v))<\text{depth}(i)$。

8.5.2 算法的非形式化描述

算法 8.8　BRIDGES ALGORITHM

输入：无向连通图$G(V,E)$, $|V|=n$的邻接矩阵；

输出：G的所有桥。

begin

(1) 构造$G(V,E)$的一棵逆树$T(V,E')$；

(2) 对图G的每个顶点$v\in V$，计算HLCA(u)；

(3) 对每个顶点$v\in V$，计算$\alpha(v)$，其中：

$\alpha(v)=\min\{\text{depth}(\text{HLCA}(w))\mid v\leqslant w$ 在T中$\}$；

(4) 检查每条树边$<i,F(i)>\in E'$，若边$<i,F(i)>$是桥，当且仅当$\text{depth}(i)\leqslant\alpha(i)$

end.

定理 8.10　在 SIMD－CREW PRAM 上，计算一个无向连通图 $G(V,E)$, $|V|=n$ 的全部桥，算法 8.8 需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器。

证明　算法 8.8 的第 (1) 步可应用第四章的最优连通分支算法及 Sollin 算法求得。由定理 4.5 及推论 5.1 知，这一步需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器；算法的第 (2) 步由定理 8.3 知，需 $O(n/K+\log n)$ 时间、$O(nK)$ 处理器 ($K\geqslant 1$)；第 (3) 步实现如下：根据第 (2) 步已算出的矩阵 F^+ 及向量 depth，对每个 j，搜索 F^+ 的第 $n-1$ 列到第 depth(j) 列，若 j 在第 i 行出现，则 i 是 j 的一个子孙，故这一步计算需 $O(n/K+\log K)$ 时间、$O(nK)$ 处理器；最后一步需 $O(1)$ 时间、$O(n)$ 处理器。故整个算法需 $O(n/K+\log^2 n)$ 时间、$O(nK)$ 处理器 ($K\geqslant 1$)。

另一种求桥的方法是：可利用图的双连通分支算法直接地求得桥，由单条边组成的双连通分支的那条边都是桥。由于篇幅所限，在此不再介绍了。

8.5.3　二维网孔上的桥算法

Atallah 等人[4] 曾给出二维网孔上的桥算法。设图 $G(V,E)$, $|V|=n$ 是一个无向连通图。设二维网孔上有 n^2 个处理器，令在位置 (i,j) 上的处理器为 PE_{ij}。在处理器 PE_{ij} 中，若终止时的 $\text{flag}(i,j)=1$，则图 G 的边 (i,j) 是一座桥 ($1\leqslant i,j\leqslant n$)。

令 $S_k(i,j)$ 表示在 G 中顶点 i 到顶点 j 的最短路径长度，且这条最短路径至多经过 k 个中间顶点。若 i 和 j 之间不存在一条有限长度的路径，则 $S_k(i,j)=\infty$，$1\leqslant i,j,k\leqslant n$。令

$$S_0(i,j)=\begin{cases}1, & 若\ (i,j)\in E\\ 0, & 若\ i=j\\ \infty, & 否则\end{cases}$$

引理 8.10　在 SIMD 二维网孔模型上，计算一个无向连通图 $G(V,E)$, $|V|=n$ 的宽度优先生成树，需要 $O(n)$ 时间、$O(n^2)$ 处理器。

证明　不失一般性，假定 $V=\{1,2,\cdots,n\}$，顶点 1 作为树根。首先计算从根出发到其它

所有顶点 $j \in V$ 的最短路径 $S_n(1,j)$. 由定理 6.4 可知，这一步需 $O(n)$ 时间、$O(n^2)$ 处理器；其次，令 $\text{level}(i) = S_n(1,i)$, $(S_n(1,i) = S_n(i,1))$，每个处理器 PE_{ij} 做纵向广播，将 $\text{level}(j)$ 的值送入第 j 列的全部处理器 PE_{ij} 中 $(1 \leqslant i \leqslant n)$. 类似地，每个 PE_{i1} 把 $\text{level}(i)$ 广播到第 i 行的所有处理器 PE_{ij} 中 $(1 \leqslant j \leqslant n)$。这样做的结果，每个处理器 PE_{ij} 里含有 $\text{level}(i)$ 及 $\text{level}(j)$ 的值；第三，我们定义：$F(i) = \min\{j \mid (i,j) \in E$ 且 $\text{level}(i) = \text{level}(j) + 1\}$, $i \neq 1$. 接着计算所有顶点 $i \in V$ 的 $F(i)$。完成以上的每一步，均需 $O(n)$ 时间、$O(n^2)$ 处理器. 这样，我们获得一个有向宽度优先生成树 $T(V,E')$。最后一步，计算和 $T(V,E')$ 对应的无向图 $T'(V,E_1)$ 的邻接矩阵. 具体实现是：假设算出的 $F(i)$ 存放在 PE_{i1} 中。1) 每行处理器做横向广播，将 $F(i)$ 送入每个处理器 PE_{ij} 中 $(1 \leqslant j \leqslant n)$; 2) 每列的处理器做纵向广播，将 $F(j)$ 送入每列的处理器 PE_{ij} 中 $(1 \leqslant i \leqslant n)$; 3) 当且仅当 $j = F(i)$ 或 $i = F(j)$ 时，则处理器 PE_{ij} 置 $T'(i,j) = 1$. 实现了上述各步后，就获得了 G 的一个以顶点 1 为根结点的无向宽度优先生成树 $T'(V,E_1)$。又因每步需 $O(n)$ 时间、$O(n^2)$ 处理器，故整个过程需 $O(n)$ 时间、$O(n^2)$ 处理器.

令 G^* 是 G 的传递闭包。定义集合 $R_G(i)$ 为：$R_G(i) = \{j \mid$ 在 G 中存在 i 到 j 的路径$\}$。

引理 8.11 令 $T(V,E')$是 G 的一棵逆树，$T'(V,E_1)$是 T'对应的无向图，$(i,j) \in E_1$，图 $G' = (G-T') \oplus T$，若边(i,j)是 G 的桥，当且仅当 $R_T(j) = R_{G'}(j)$.

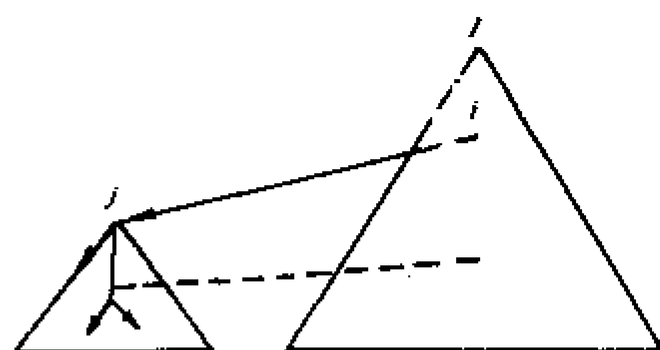


图 8.7 存在桥时，逆树中顶点和非树顶点的连接

证明 若 $i = F(j)$且$(i,j) \in E_1$ 是 G 的一座桥，则在边集合 $E-E_1$ 中不存在这样的边，它将 j 的子树顶点(包括 j)同 j的非子树顶点连接起来，(如图 8.7 的虚线所示)。故有 $R_T(j) = R_{G'}(j)$.

如果 $i = F(j)$且 $R_T(j) = R_{G'}(j)$，那么在 $E-E_1$ 中不存在一条连接 j的子树顶点到 i 的子树之外的顶点的边。故当$(i,j) \in E$ 从 G 中删除后，将使 G 不再是连通的. 所以边(i,j)是 G 的一座桥.

有了上面的概念及引理，现在我们介绍关于桥的算法。

算法 8.9 BRIDGES ALGORITHM ON MESH ARRAY

输入：无向图$G(V,E)$的邻接矩阵A. 处理器PE_{ij}的单元$A(i,j)$存贮矩阵A的一个元素；

输出：图G的全部桥。若$\text{flag}(i,j) = 1$，则处理器PE_{ij}标明边(i,j)是桥。

begin

(1) 计算无向连通图$G(V,E)$的有向宽度优先生成树$T(V,E')$及对应的无向树$T'(V,E_1)$的邻接矩阵；

(2) 构造子图$G'(V,\hat{E})$ 其中$\hat{E} = E' \oplus (E-E_1)$;

(3) 计算G'及T的传递闭包G'^*及T^*；

(4) 对每个$j(\neq 1)$，先判断邻接矩阵T^*的第j行和邻接矩阵G'^*的第j行是否相等？然后通过水平旋转收集“相等”和“不相等”的信息存放在PE_{jj}中。若相等，则PE_{jj}送出标志$PE_{F(j),j}$及$PE_{j,F(j)}$的flag为1。即$(F(j), j)\in E_1$是G的桥

end .

定理 8.11 在 SIMD 二维网孔模型上，计算一个无向连通图 $G(V,E)$，$|V| = n$ 的所有桥，算法 8.9 需 $O(n)$ 时间、$O(n^2)$ 处理器。

证明 算法 8.9 的第 (1) 步，根据定理 6.4，需 $O(n)$ 时间、$O(n^2)$ 处理器；算法的第 (2) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (3) 步由定理 6.4 知，亦需 $O(n)$ 时间、$O(n^2)$ 处理器；第 (4) 步广播需 $O(n)$ 时间、$O(n^2)$ 处理器。标识桥需 $O(n)$ 时间、$O(n^2)$ 处理器。故整个算法需 $O(n)$ 时间、$O(n^2)$ 处理器。

8.6 双连通分支算法的应用——求图的关节点

前面我们描述了求图的双连通分支算法 8.3，现在介绍如何从求得的双连通分支来决定图 $G(V,E)$ 的关节点集。众所周知，如果顶点 a 不是 G 的生成树 T 的根 r，那么，a 是 G 的一个关节点，当且仅当对某一个 j 来讲，B_j 是 G 的一个双连通分支且 a 是 $T\cap B_j$ 的根顶点；树 T 的根是 G 的关节点，当且仅当 r 至少是两个不同的子树 $T\cap B'$ 和 $T\cap B''$ 的根，其中 B' 和 B'' 分别是 G 的两个不同的双连通分支。

利用上述的观察，Tsin 等人曾建议一个计算 G 的关节点集算法。限于篇幅，这里就不详细介绍了。下面定理的证明，亦概括了算法的主要过程。

定理 8.12 在 SIMD－CREW PRAM 上，求一个无向连通图 $G(V,E)$，$|V| = n$ 的关节点集，需 $O(n/K + \log^2 n)$ 时间、$O(nK)$ 处理器 $(K \geq 1)$。

证明 首先构造 G 的一棵逆树 $T(V,E')$；其次确定 G 的双连通分支的顶点集，这一步根据定理 8.5，需 $O(n/K + \log^2 n)$ 时间、$O(nK)$ 处理器 $(K \geq 1)$；第三，确定每条有向边 $e = \langle u,v \rangle \in E'$ 的头 head(E)（即 head(e) $= v$），这一步需 $O(1)$ 时间、$O(n)$ 处理器；第四，将所有头 (head(e)) 顶点集合分组，按照它们所在的双连通分支顶点集来分组，这一步可通过排序解决，而排序需 $O(\log n)$ 时间、$O(n)$ 处理器；最后，每组头顶点中 depth 值最小的顶点，即是关节点。树根是关节点，当且仅当它被两组或两组以上的头顶点选作为 depth 值最小的顶点，完成这一步需 $O(\log n)$ 时间、$O(n)$ 处理器。所以整个过程需 $O(n/K + \log^2 n)$ 时间、$O(nK)$ 处理器 $(K \geq 1)$。

8.7 小　结

在并行环境中，怎样迅速地解决无向图的一些基本性质问题，如基本回路、关节点、双连通分支和桥等，是一个十分重要的问题。本章着重介绍了它们的一些并行算法。主要内容有：基于 SIMD 共享存贮模型叙述了一个求基本回路的算法；给出了两个采用不同策略的求图的双连通分支算法；基于 SIMD 共享存贮模型和 SIMD 二维网孔模型，分别介绍了计算图的所有桥的算法；最后讨论了怎样利用图的双连通分支去计算图的关节点。

研究图的一些基本性质问题的并行算法，最早始于 Savage[1]，基于 SIMD－CREW PRAM 模型，她和她的同事利用图的传递闭包方法，曾给出了 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器的求基本回路的算法，和 $O(\log^2 n)$ 时间、$O(n^3/\log n)$ 处理器的求双连通分支算法，以及 $O(\log^2 n)$ 时间、$O(n^2\log n)$ 处理器的计算桥的并行算法[1,2]。Tarjan 等人在 SIMD－CRCW PRAM 上给出求双连通分支算法后，还建议了一个在 SIMD－CREW PRAM 上的实现算法，该算法需 $O(n^2/p)$ 时间、$O(p)$ 处理器 ($1\leqslant p\leqslant n^2/\log^2 n$)[6]。Atallah 等人在二维网孔模型上，还叙述了一个 $O(n)$ 时间、$O(n^2)$ 处理器的求关节点算法[4]。Huang 曾在树网上给出一个 $O(n^2/p)$ 时间、$O(p)$ 处理器的求最优双连通分支算法 ($1\leqslant p\leqslant n^2/\log^2 n$)[9]。最近，Tsin 基于 SIMD－CREW PRAM 模型，建议了两个计算树中每对顶点的最低公共祖先算法，该算法可计算 r 对顶点的最低公共祖先，分别需 $O((n+r/p)\log n)$ 时间及 $O(n^2/p+\log n)$ 处理器，尤其是当 $1\leqslant p\leqslant n^2/\log^2 n$ 时，利用求最低公共祖先算法，可推得一个最优的求基本回路算法[10]。此外，Goshi 等人基于一维线性阵列，给出了最优的计算桥及关节点算法，算法需 $O(n^2/p)$ 时间、$O(p)$ 处理器 ($1\leqslant p\leqslant n$)[11]。

有关这些问题的并行算法不胜枚举，这里介绍的内容，基本上包括当前其它算法的主要思想，由于篇幅缘故，在此不再叙述了。

参 考 文 献

[1] Savage C. Parallel Algorithms for Graph Theoretic Problems, Ph.D diss. Dept. Math., Univ. Illinois, Urbana, 1977

[2] Savag C, Ja′ Ja′ J. Fast Efficient Parallel Algorithms for Some Graph Problems, *SIAM J. Comput.*, 10(4), 1981, 682–691

[3] Tsin Y H, Chin F. Y. Efficient Parallel Algorithms for a Class of Graph Theoretic Problems, *SIAM J. Comput.*, 13(3), 1984, 580–599

[4] Atallah M J, Kosaraju S R. Graph Problems on a Mesh–Connected Proceesor Array, *J. ACM*,

31(3), 1984, 649–667

[5] Tarjan R E, Vishkin U. Finding Biconnected Components and Computmg Tree Functions in Logarithmic Parallel time, *Proc. 25th Annual IEEE Sympo. on FOCS*, 1984, 12–20

[6] Tarjan R E, Vishkin U. An Efficient Parallel Biconnectivity Algorithm, *SIAM J. Comput.*, **14**(4), 1985, 862–874

[7] Cole R. Parallel Merge Sort, *SIAM J. Comput.*, **17**(4), 1988, 770–785

[8] Shiloach Y, Vishkin U. An O(log*n*) Parallel Connectivity Algorithm, *J. Algorithms*. **3**, 1982, 57–63

[9] Huang D M. Solving Some Graph Prblems with Optimal or Near–Optimal Speedup on Mesh–of–Trees Networks, *Proc. 26th Annual IEEE Sympo. on FOCS*, 1985, 232–240

[10] Tsin Y H. Finding Lowest Common Ancestors in Parallel, *IEEE Trans. Comput.*, C–**35**(8), 1986, 764–769

[11] Goshi K A, Varman P J. Optimal Graph Algorithms on a Fixed–Size Linear Array, *IEEE Trans. Comput.*, C–**36**(4), 1987, 460–470

第九章　欧拉图及哈密顿图的并行算法

9.1 找欧拉回路的算法

在一个欧拉图中找欧拉回路的问题是一个非常古老的问题。在单机系统上人们已经设计了许多有效的算法。但大多数算法基本上都是基于同一思想：即把图的边集合划分为许多回路，然后将那些没有公共边的回路“缝合”起来组成原来欧拉图的**欧拉回路**(Euler Cycle)。在串行算法中，这些回路是一条接着一条顺序发现的，因而缝合也是顺序进行的。然而在并行环境里，虽然没有公共边的回路可以并行地产生，但没有一种明显的办法将所有的回路并行地缝合起来，并确信缝合后的整条回路就是欧拉回路。Awerbuch 等人及 Atallah 等人巧妙利用了找图的连通分支及 MST 并行算法，分别在 SIMD－CRCW PRAM 上给出了一个 $O(\log n)$ 时间、$O(m+n)$ 处理器的找欧拉回路的并行算法[1,2]。本节我们将详细介绍 Atallah 的算法。

9.1.1　欧拉图的基本概念及性质

设 $G(V,E)$，$|V|=n$，$|E|=m$ 是一个无向图。若在 G 中存在一条由所有的边组成的路径，且 G 的每条边在这条路径上仅出现一次，则这条路径是图 G 的一条**欧拉路径**(Euler Path)。若欧拉路径是一条首尾相连的路径，则这条路径是图 G 的一条**欧拉回路**。一个含欧拉回路的图称为**欧拉图**(Euler Graph)。设 $G(V,E)$ 是一个有向图，若存在一条包含所有边的路径，且每条有向边在这条有向路径上仅出现一次，则这条有向路径是图 G 的一条**有向欧拉路径**。若有向欧拉路径形成一条有向闭合路径，则这条有向闭合路径是图 G 的一条**有向欧拉回路**。具有有向欧拉回路的图称为**有向欧拉图**。

欧拉图有许多有趣的性质。这里仅给出一些重要的基本性质。

引理 9.1　(1) 无向图 $G(V,E)$ 含有一条欧拉路径，当且仅当 G 是连通的，且 G 仅有两个顶点度数为奇数，这条欧拉路径是从这两个奇数度顶点的其中一个开始到另一个结束。(2) 无向图 $G(V,E)$ 含有一条欧拉回路，当且仅当 G 是连通的，而且 G 的所有顶点度数均为偶数。

证明　必要性。设图 G 含有欧拉路径，即有顶点及边序列 $v_0e_1v_1e_2\cdots e_iv_ie_{i+1}\cdots e_kv_k$，其中 $e_j\in E$，$v_i\in V$，$1\leqslant i<n$，$0\leqslant j<m$。根据欧拉路径定义，在上述序列中的顶点可以重复出现，但边不允许重复。因欧拉路径经过了图的所有顶点。故 G 是连通的。

在欧拉路径中，对任意一个非路径端点的顶点 v_i，每当它出现一次，必关联着两条边。故 v_i 虽可重复出现，但其度数 $d(v_i)$ 必是偶数。对于端点而言，若 $v_0=v_k$，则 $d(v_0)$

为偶数，即 G 中无奇数度顶点，这条欧拉路径是 G 的一条欧拉回路．若 $v_0 \neq v_k$，则 $d(v_0)$ 及 $d(v_k)$ 均为奇数，G 中就有两个奇数度顶点，这条欧拉路径始于奇数度顶点，最后也终止于奇数度顶点。

充分性．若图 G 是连通的，且有两个奇数度顶点或全为偶数度顶点．构造 G 的一条路径如下:

(1) 若有两个奇数度顶点，则从其中的一个顶点开始构造一条路径，即从 v_0 出发经关联边 e_1 进入 v_1。若 $d(v_1)$ 为偶数，则必可由 v_1 再经关联边 e_2 进入 v_2。如此进行下去，每边仅取一次．由于 G 是连通的，故可到达另一奇数度顶点后停下来，得到一条路径 $v_0e_1v_1e_2\cdots v_ie_{i+1}\cdots v_k$。若 G 的顶点度数均为偶数，则从任一顶点 v_0 出发，用上述方法必可回到顶点 v_0，得到一条回路 L_1。

(2) 若L_1包含了G的所有边，则L_1就是G的一条欧拉回路。

(3) 若 G 中去掉 L_1 后得到子图 G'，则 G' 中每个顶点的度数为偶数。又因为原来 G 是连通的，故 L_1 与 G' 至少有一个顶点 v_i 重合，在 G' 中从 v_i 出发重复 (1) 的方法，得到闭合回路 L_2。

(4) 若 $L_1 \cup L_2$ 包含了 G 的所有边，则它就是 G 的一条欧拉回路（路径）。否则重复 (3) 可以得到回路 L_3，由此类推直到得到一条经过图 G 所有边的欧拉回路（路径）．

对有向图 $G(V,E)$，我们用 $d_{in}(v)$ 及 $d_{out}(v)$ 分别表示顶点 $v\in V$ 的入度和出度。它也有一个重要的基本性质。

引理 9.2 (1) 有向图 $G(V,E)$ 存在一条有向欧拉路径，当且仅当存在两个特殊的顶点 u，v，满足：$d_{out}(v)+1=d_{in}(v)$，$d_{out}(u)=d_{in}(u)+1$，而其它所有的顶点 w 有 $d_{out}(w)=d_{in}(w)$，且这条欧拉路径始于 u 终于 v，$u\in V$，$v\in V$，$w\in V$。(2) 有向图 $G(V,E)$ 存在一条欧拉回路，当且仅当对任意一个顶点 $v\in V$ 均有 $d_{in}(v)=d_{out}(v)$。

证明 类似于引理 9.1 的证明．事实上，可看作这是对无向图的欧拉路径（回路）的推广。对于有向图的任意一个顶点 $v\in V$ 来讲，若 $d_{in}(v)=d_{out}(v)$，则该顶点的度数总是偶数。若 $d_{in}(v)+1=d_{out}(v)$ 或者 $d_{out}(v)+1=d_{in}(v)$，则该顶点的度数总是奇数．

9.1.2 找有向欧拉回路的算法

无论是有向图还是无向图，这里仅讨论算法的输入是一个有向欧拉图 $G(V,E)$，$|V|=n$，$|E|=m$。这样做的原因有两条：其一是对任何一个输入的图 $G(V,E)$，我们可根据欧拉图的性质对之进行检测，以确定 G 是否是欧拉图．在 SIMD－CRCW PRAM 上，若图用邻接表形式表示，计算每个顶点的度数（无向图）和入度、出度（有向图），只需 $O(\log n)$ 时间、$O(n+m)$ 处理器；计算奇数度顶点个数可在 $O(\log n)$ 时间完成，且只需 $O(n)$ 处理器．检查 G 的连通性仅需 $O(\log n)$ 时间、$O(n+m)$ 处理器[3]。其二，对一个

具有欧拉路径的图 $G(V,E)$ 来讲，通过在两个奇数度顶点之间加一条（有向）边，使得 $G(V,E)$ 变为 $G''(V,E'')$ 的欧拉图。对有向图来讲，若 $d_{in}(u)=d_{out}(u)+1$ 且 $d_{out}(v)=d_{in}(v)+1$，则在图中增加边 $<u,v>$。计算出 G'' 的有向欧拉回路后去掉回路上边 $<u,v>$ 就得出图 G 的有向欧拉路径 $(u\in V, v\in V)$。令 C_x 是一个有向回路或无向回路，我们用 $V(C_x)$ 表示回路 C_x 上的顶点集合。

1. 算法的基本原理

若欧拉图 $G(V,E)$ 的边集合 E 被划分成许多没有公共边的回路 C_1，C_2，…，C_i，…，$C_k(k\leqslant m)$，且每条边 $e\in E$ 仅出现在唯一的一条回路上，但不同的回路可有相同的顶点。这种划分我们称之为欧拉划分。若对每条边 $e\in E$，指定唯一的一条后继边，使得没有两条边将同一条边作为它们的后继边，则这种欧拉划分是唯一确定的划分。边 $e=<u,v>$ 的后继边 SUCC(e) 可以是 v 的任何一条射出边。显然 e 就变成 SUCC(e) 的前趋边。

Atallah 算法的基本思想是：首先对一有向欧拉图进行欧拉划分，设划分后的回路集合 $\Psi=\{C_1,\cdots,C_k\}$，使得 $C_i\cap C_j=\phi\ (i\neq j)$ 且 $\bigcup_{i=1}^{k} C_i=E$。然后构造一个由划分确定的辅图 $G_1(V_1, E_1)$（在后面将给出定义），G_1 是一个无向连通二分图；在 G_1 上构造一棵生成树 T；对 T 的每条边定义两个方向，结果变为一个有向图 $\tilde{T}$，$\tilde{T}$ 一定是一个有向欧拉图。找出 $\tilde{T}$ 的欧拉回路 $C_{\tilde{T}}$；最后利用 $C_{\tilde{T}}$ 将 Ψ 的所有回路缝合起来就得到 G 的有向欧拉回路。

引理 9.3 设 $T(V,E_T)$，$|V|=n$，$|E_T|=n-1$ 是任意一棵树。若对每条边定向，使得每条无向边变成两条方向相反的并行有向边，则得到的图 $\tilde{T}(V, E_{\tilde{T}})$ 是一个有向欧拉图，这里 $E_{\tilde{T}}=\{<u,v>|(v,u)\in E_T\}\cup\{<v,u>|(v,u)\in E_T\}$。

证明 设树 T 的任一顶点 $v\in V$ 的度数为 $d(v)$，则在 $\tilde{T}$ 中顶点 v 的入度 $d_{in}(v)=d(v)$，出度 $d_{out}(v)=d(v)$，即 $d_{in}(v)=d_{out}(v)$。又 T 是连通的，故 $\tilde{T}$ 必然也是连通的。所以 $\tilde{T}$ 是一个有向欧拉图。

2. 算法的非形式化描述

算法9.1 FINDING DIRECTED EULER CYCLES

输入：有向图 $G(V,E)$，用邻接表形式存贮，$V=\{1, 2, \cdots, n\}$；

输出：数组SUCC(1:m)，其中第 i 条边的后继边SUCC($i+1$)，$(i\leqslant m-1)$，第 m 条边的后继边为SUCC(1)。

begin

(1) 将图 $G(V,E)$ 的边按欧拉划分划分成许多无公共边的回路集合

$$\Psi=\{C_1, C_2, \cdots, C_k\};$$

(2) 构造辅图$G_1(V_1,E_1)$，它是一个无向二分图。其中$V_1 = X \cup Y$，X是“回路顶点”集．Y是G的顶点集即$Y = V$。每一条回路C_x是G_1的一个回路顶点$(1 \leqslant x \leqslant k)$．边集合$E_1$定义为：

$$E_1 = \{(u,\ C_x) |\ u \in V,\ C_x \in \Psi,\ u \in V(C_x),\ 1 \leqslant x \leqslant k\};$$

(3) 找出G_1的生成树$T(V_1, E_T)$，把树T的每条边赋予两个方向后变成方向相反的一对并行边，结果得到一个有向欧拉图$\tilde{T}(V_1, E_{\tilde{T}})$；

(4) 找出$\tilde{T}$的欧拉回路，然后以这个回路为基础，将C_1，…，C_k缝合起来得到G的欧拉回路

end .

在进一步描述算法的实现细节之前，首先介绍 Tarjan 等人用树 $T(V_1, E_T)$ 导出有向图$\tilde{T}(V_1,E_{\tilde{T}})$ 的欧拉回路算法[4]．他们将得出的欧拉回路用一个含有 $2|E_T|$ 元素的一维数组 FOLLOW 存贮，对 T 中任意一个顶点 $v \in V_1$，设 $d(v) = d$，v 在 T 中的 d 个关联边分别为 (v, u_0)，…，$(v,\ u_{d-1})$，$u_i \in V$，$i = 0$，…，$d-1$．对任一顶点 v，算法执行 $\mathrm{FOLLOW}(<u_i, v>) \leftarrow \mathrm{FOLLOW}(<v, u_{(i+1) \bmod d}>)$，$0 \leqslant i \leqslant d-1$．这样，数组 FOLLOW 给出了$\tilde{T}$ 的每条边的后继边．因此，若已知$\tilde{T}$ 的一条边，则 FOLLOW 将给出从这条边开始的一条路径。事实上，FOLLOW 给出了$\tilde{T}$ 的一条有向欧拉回路。

下面我们给出在 SIMD－CRCW PRAM 计算模型上，找有向欧拉回路的算法 9.1 中各步的实现．

算法 9.1 的第 (1) 步由两个子步组成．

Step1.1　对所有边 $<i,j> \in E$ 按字典次序定序。任意两条边 $<a,b>$ 和 $<c,d>$，$<a,b>$ 的字典序先于 $<c,d>$ 当且仅当 $a<c$ 或 $a=c$ 且 $b<d$。令 OUT_EDGE(v,k) 是顶点 v 的第 k 条射出边 $(1 \leqslant k \leqslant d_{out}(v))$．对 $<i,j> \in E$ 的边按逆字典序排序，即按 $<j,i>$ 排序．令 IN_EDGE(v,k) 是顶点 v 的第 k 条入边 $(1 \leqslant k \leqslant d_{in}(v))$。

Step1.2　**for each** v, k: $(1 \leqslant v \leqslant n)$ **and** $(1 \leqslant k \leqslant d_{in}(v))$　**pardo**

　　SUCC(IN_EDGE(v,k)) ← OUT_EDGE(v,k);

　　NEXT(IN_EDGE(v,k)) ← OUT_EDGE(v,k)

endfor ;

step1.1 可应用 Cole 的排序算法来实现，需 $O(\log m)$ 时间、$O(m)$ 处理器[6]；Step1.2 将 G 的边集合划分成许多没有公共边的回路。数组 SUCC 和辅助数组 NEXT 对 G 的每条边定义唯一的一条后继边．最终的目标将使数组 SUCC 含有欧拉回路。辅助数组 NEXT 是数组 SUCC 的一个拷贝，在后面计算中要用到它．Step1.2 只需 $O(1)$ 时间、$O(m)$ 处理器；所以，第 (1) 步需 $O(\log m)$ 时间、$O(m)$ 处理器．

算法 9.1 的第 (2) 步由 Step2.1 及 Step2.2 组成。

Step2.1　将每条回路 C 上字典序最小的边选作回路在辅图 G_1 上的回路顶点，它代表回路 C。为此，用一维数组 D 来标识边所在的回路。对任一边 $<i,j> \in E$，$D(<i,j>)$ 初始化时为 $<i,j>$。最终 $D(<i,j>)$ 等于边 $<i,j>$ 所在回路上最小字典序的边，即具有相同 D 值的所有边位于同一回路上，具体实现是:

```
for k ← 1 to ⌈logm⌉ do
    for each e = <u, v> : e ∈E pardo
        D(e) ← min{D(e), D(NEXT(e)};
        NEXT(e) ← NEXT(NEXT(e))
    endfor
endfor ;
```

若边 e 满足 $D(e)=e$，则 e 是它所在回路的标识。

Step2.2　构造一个辅图 $G_1(V_1,E_1)$，G_1 是一个无向二分图，其中 $V_1=\{v \mid v\in V$ 或它是由 SUCC 定义的一条回路 $\}$；$E_1=\{(v,C_i) \mid C_i$ 是回路顶点，$C_i\in\Psi, v\in V(C_i)\}$。注意：一个顶点 v 可能在一条回路 C_i 上出现多次，每次出现时，v 是通过 $<i,v> \in E$ 的边进入的。令 CERTIFICATE$((v,C_i))$ 是回路 C_i 上一条形式为 $<i,v> \in E$ 的边。

Step2.1 只需 $\lceil \log m \rceil$ 次循环就足够了。因为一条回路最多只有 $O(m)$ 条边，故这一子步需 $O(\log m)$ 时间、$O(m)$ 处理器；Step2.2 可应用排序算法产生图 G_1 和选择出 CERTIFICATE$((v,C_i))$，$(v,C_i)\in E_1$。因 G_1 的顶点数至多为 $O(n+m)$，边数至多为 $O(n+m)$，所以这一子步需 $O(\log n)$ 时间、$O(n+m)$ 处理器。因而整个第 (2) 步需 $O(\log m)$ 时间、$O(n+m)$ 处理器。

算法 9.1 的第 (3) 步由 Step3.1 和 Step3.2 组成。

Step3.1　构造 G_1 的生成树 $T=(V_1,E_T)$；

Step3.2　将树 T 的每条边 $e=(v,u)\in E_T$ 替换成方向相反的一对并行边 $<u,v>$ 及 $<v,u>$，从而构成有向欧拉图 $\tilde{T}(V_1, E_{\tilde{T}})$。

Step3.1 应用 Shiloach 等人在 SIMD – CRCW PRAM 模型上的求连通分支算法[3]，这一子步需 $O(\log m)$ 时间、$O(n+m)$ 处理器。Step3.2 在使用 $O(n+m)$ 个处理器情况下，仅需 $O(1)$ 时间就可完成。故第 (3) 步需 $O(\log m)$ 时间、$O(n+m)$ 处理器。

算法 9.1 的第 (4) 步亦由两个子步组成。

Step4.1 的目标是将数组 SUCC 变成更大的回路，使得回路上的边是 G 及 T 的边交替出现。且回路上的边要满足两条性质：其一，$\tilde{T}$ 的边在回路上出现的次序同它在 $\tilde{T}$ 的欧拉回路上出现的次序一致；其二，G 的边在回路上出现的次序同它在 G 的欧拉回路出现

的次序一致．因为性质二由性质一很容易得到，所以算法的关键是给出具有性质一的回路。为了便于计算，我们引入一个更大的数组 SUCC1，存贮由 G 及 $\bar{T}$ 边交替组成的回路．初始化时，对任意一条边 $e \in E$，做：$\mathrm{SUCC1}(e) \leftarrow \mathrm{SUCC}(e)$

对顶点 $v \in V_1$ 且 $v \in V$，设在树 T 中 $d(v)=d$，v 的 d 条关联边分别为 (v,u_0)，…，(v,u_{d-1})，那么在回路中 v 的后继者由

$$\mathrm{SUCC1}(<u_i,v>) \leftarrow \mathrm{SUCC1}(<v_1,u_{(i+1) \bmod d}>),\ 0 \leqslant i \leqslant d-1$$

确定对每条回路顶点 $C_i \in \Psi$ 且 $C_i \in V_1$，令 C_i 在 T 中度数 $d(C_i)=d$，它的 d 条关联边为 (v_0,C_i)，…，(v_{d-1}, C_i)．注意：$v_i \in V$，$0 \leqslant i \leqslant d-1$．令 $\mathrm{CERTIFICATE}((v_\alpha,C_i))$ 是 $<i_\alpha,v_\alpha>$，$0 \leqslant \alpha \leqslant d-1$．这 d 条证实边 (Certificate Edge) 都是 G 的边，它们将按一定的次序出现在回路 C_i 上。这一子步的基本思想是：从边 $e \in E$ 在 C_i 上的次序可获得 e 在 $\bar{T}$ 的回路上的次序．注意在算法 9.1 第 (1) 步中．已有

$$\mathrm{SUCC}(<i_\alpha,\ v_\alpha>) = <v_\alpha,\ j_\alpha>,\ 0 \leqslant \alpha \leqslant d-1,$$

也就是说，在 C_i 上 $<v_\alpha,\ j_\alpha>$ 是 $<i_\alpha,v_\alpha>$ 的后继边。

step4.1 **for** each $e: (e=(v_\alpha,C_i) \in E_T) \wedge (0 \leqslant \alpha \leqslant d-1)$ **pardo**

$e_1 \leftarrow \mathrm{CERTIFICATE}((v_\alpha,C_i))$;

$\mathrm{SUCC1}(<V_\alpha,\ C_i>) \leftarrow \mathrm{SUCC}(e_1)$;

$\mathrm{SUCC1}(e_1) \leftarrow <C_i,\ v_\alpha>$

endfor;

下面证明：完成 Step4.1 后，性质二必然成立．

引理 9.4 算法 9.1 的第 4.1 步结束时，SUCC1 含有 G 的一条欧拉回路．

证明 设 $<i,j> \in E$，我们证明它在由 SUCC1 定义的回路上。算法在执行了第 (1) 步后，$<i,j>$ 在某条回路 C_x 上 $(1 \leqslant x \leqslant k)$．若在 G_1 的生成树中，回路顶点 C_x 的度数为 d，则回路 C_x 含有 d 条证实边将 C_x 划分成 d 条路径：每条路径开始于证实边的后继边，终止于下一条证实边．$<i,j>$ 必须位于某一条路径上．而每条证实边必须在 SUCC1 定义的回路上。因此 $<i,j>$ 必位于 SUCC1 定义的回路上．而且第二次出现这条路径时必须在遍历了 G 的所有其它边之后，否则根据性质一，与 SUCC1 定义相矛盾．

Step4.2 **for** each e: $e \in E \cup E_{\bar{T}}$ **pardo**

if $\mathrm{SUCC1}\ (e) \in E_{\bar{T}}$ **then** $\mathrm{SUCC1}(e) \leftarrow \mathrm{SUCC1}(\mathrm{SUCC1}(e))$ **endif**

endfor;

for each $e: e \in E$ **pardo**

$\mathrm{SUCC}(e) \leftarrow \mathrm{SUCC1}(e)$

endfor ;

Step4.1 需 $O(1)$ 时间、$O(n+m)$ 处理器；**Step4.2** 是从 SUCC 定义的回路上消除辅图 G_1 后导出的欧拉图 $\tilde{T}$ 的边。由 **Step4.1** 可知，不存在两条同属于 $\tilde{T}$ 的边作为前趋和后继，故 **Step4.2** 也只需 $O(1)$ 时间、$O(n+m)$ 处理器。因此，整个第（4）步需 $O(1)$ 时间、$O(n+m)$ 处理器。

定理 9.1 在 SIMD－CRCW PRAM 上，求一个有向欧拉图 $G(V,E)$，$|V|=n$，$|E|=m$ 的欧拉回路，算法 9.1 需 $O(\log n)$ 时间、$O(n+m)$ 处理器。

证明 通过上面对算法各步的分析，容易看出。

对于找无向图 $G(V,E)$ 的欧拉回路，我们可采用类似的方法。也就是将 $G(V,E)$ 的边给予定向，使之成为一个有向图，然后利用有向图的求欧拉回路算法求出 G 的欧拉回路（抹去边的方向）。

令 $G(V,E)$ 是一个无向图，而且是一个欧拉图。令 G' 是将 G 的每条边 $(u,v)\in E$ 定向为方向相反的两条有向并行边 $<u,v>$ 及 $<v,u>$ 后的有向图。在 G' 中究竟两条有向边中的哪一条将代表 G 中相应的边呢？现在来研究它。令 $v\in V$ 的度数为 d。因 G 是欧拉图，故 d 为偶数。令 (v, u_1)，(v, u_2)，…，(v, u_d) 是 v 的关联边。按下述方式对有向图 G' 进行欧拉划分。对每个奇数 i: $1\leqslant i<d$ 做：

$$\text{NEXT}(<u_i, v>)\leftarrow <v, u_{i+1}>;$$

$$\text{NEXT}(<u_{i+1}, v>)\leftarrow <v, u_i>;$$

将数组 NEXT 定义的许多无公共边的回路成对地进行划分。对每条回路 C_x 来讲，存在另一对偶回路 C'_x，C'_x 是由 C 的所有相反方向的边组成的。我们的目标是从 G' 的每对回路中删除其中的一条回路，最后每对方向相反的边只剩下一条，这条有向边就代表 G 的对应边。

对有向图 G' 的每条边 $e=<i,j>$，计算它所在回路上字典序最小的边 $D(e)$，然后，对 G 的每条边 $e=(u,v)\in E$ 来讲，若 $D(<u,v>)$ 的字典序号小于 $D(<v,u>)$ 的字典序号，则把边 $<v,u>$ 从 G' 中删去。由于 G 满足引理 9.1，因而从 G' 中删除一些边后的有向图 G'' 满足引理 9.2，又因为 G 是欧拉图。所以 G'' 是有向欧拉图。对 G'' 执行找有向欧拉回路算法后就可得到 G 的一个欧拉回路（抹去有向边的方向）。

推论 9.1 在 SIMD－CRCW PRAM 上，求一个欧拉图 $G(V,E)$，$|V|=n$，$|E|=m$ 的欧拉回路，需 $O(\log n)$ 时间、$O(n+m)$ 处理器。

证明 由无向图 G 定向成为一个有向图 G' 需 $O(1)$ 时间、$O(m)$ 处理器；对 G' 实行欧拉划分需 $O(1)$ 时间、$O(m)$ 处理器；计算每条回路的字典序最小边，由找有向欧拉回路算法 9.1 的第 (2.1) 步知，需 $O(\log m)$ 时间、$O(m)$ 处理器；由 G' 导出 G'' 需 $O(1)$ 时间、$O(m)$ 处理器；由定理 9.1，求 G'' 的有向欧拉回路需 $O(\log m)$ 时间、$O(n+m)$ 处理器；又因为 G'' 的欧拉回路也就是 G 的欧拉回路。所以找一个无向欧拉图的欧拉回路，

需 $O(\log m)$ 时间、$O(n+m)$ 处理器。

推论 9.2 在 SIMD－EREW PRAM 上，求一个有向（无向）的欧拉图的欧拉回路，需 $O(\log^2 n)$ 时间、$O(n+m)$ 处理器，这里 n 是图的顶点数目，m 是图的边数。

证明 运用不同计算模型之间的通用模拟技术[7,8]，算法 9.1 在 SIMD－EREW PRAM 上实现，需 $O(\log^2 n)$ 时间、$O(n+m)$ 处理器。

9.2 在竞赛图中找哈密顿回路算法

一个**竞赛图** (Tournament Graph) 是一个有向图 $G(V,E)$，对 V 中每一对顶点 u 和 v，或者 $<u,v>\in E$，或者 $<v,u>\in E$，但这两条边不能都在 E 中。竞赛图 $G(V,E)$ 是从 n 个选手竞赛的模型抽象出来的，每个选手必须同其它所有选手都有一场竞赛，竞赛规则是每场必有输赢而不能出现平局。竞赛图一个有用的性质是：每个竞赛图导出的子图也是一个竞赛图、若 $<u,v>\in E$，我们说 u 支配 v，将这种性质定义为 $u \succ v$。由于图中边的方向可以是任意的，故支配关系并不一定是传递的。将上述的支配关系推广到顶点集合上，令 $A\subseteq V$，$B\subseteq V$，若对任意一个顶点 $v\in A$，对另外任意一个顶点 $u\in B$，均有 $v\succ u$，则称集合 A 支配集合 B（记作 $A\succ B$）显然，$A\cap B=\phi$。已知一个顶点 $v\in V$，我们对其它顶点按照同 v 的关系分为两大类：$W(v)$ 是 v 支配的顶点集合（在比赛中 v 赢），即对任意一个 $u\in W(v)$ 内均有 $v\succ u$。$L(v)$ 是支配 v 的顶点集（在比赛中 v 输），即对任意一个 $u\in L(v)$ 均有 $u\succ v$。本节使用的其它一些记号是：对一个顶点集合 $U\subseteq V$，$G(U)$ 是指在 U 上导出的子图。对一个图 G，用 $V(G)$，$E(G)$ 分别代表 G 的顶点集合及边集合。对两个图 A，B，用 $A\cup B$ 表示这两个图合并为一个图。对两条路径 P，Q，用 $P\cdot Q$ 表示它们连接起来变成一条更长的路径。

9.2.1 竞赛图的一些基本性质

竞赛图 $G(V,E)$ 有许多重要的性质。本节只给出与介绍的算法有密切联系的几个基本性质。

引理 9.5 每个竞赛图 $G(V, E)$ 都含有一条哈密顿路径。

证明 对竞赛图的顶点数目 n 进行归纳证明。当 $n=2$ 时，命题显然成立。假定对 n 个顶点的竞赛图，命题成立。现在考虑一个具有 $n+1$ 个顶点的竞赛图 $G(V,E)$。对任意一个顶点 $v\in V$，G 的导出子图 $G(V-\{v\})$ 也是一个竞赛图。由归纳假定可知，$G(V-\{v\})$ 中存在一条哈密顿路径。这条路径的顶点序列为 v_1，…，v_n，其中 $v_i\in V-\{v\}$，$i=1$，…，n。若 $v\succ u_1$，那么 v，v_1，v_2，…，v_n 这个顶点序列构成 G 的一条哈密顿路径。否则令 i 是序列中满足 $v_i\succ v$ 的最大下标。若 $i=n$，那么顶点序

列 v_1，v_2，…，v_n，v 构成 G 的一条哈密顿路径．若 $i \neq n$，则顶点序列 v_1，v_2，…，v_i，v，v_{i+1}，…，v_n 是 G 的一条哈密顿路径。故在有 $n+1$ 个顶点的竞赛图中也存在一条哈密顿路径．

不难看到，上面的证明事实上给出了一个找 G 的哈密顿路径的有效算法。但这个算法本质上是顺序执行的。为了得到找 G 的哈密顿路径的并行算法，我们需采用不同的方法．下面将介绍用分而治之策略设计的一个并行算法[9]。

一种最简单的办法是：(1) 把竞赛图分裂成顶点数相等的两个子图 G_1 和 G_2；(2) 并行地找出 G_1 的哈密顿路径 H_1 和 G_2 的 H_2；(3) 将 H_1 与 H_2 连接起来形成 G 的一条哈密顿路径 $H_1 \cdot H_2$。但这一步尚不能保证正确地找到 G 的一条哈密顿路径，因为对 H_1 及 H_2 两头的端点无法控制。

因此，为找出 G 的哈密顿路径，必须对上述简单的方法进行适当的修改．这里的关键在于对 G 中顶点的观察．令 $v \in V$，设由 $L(v)$ 导出的子图 $G(L(v))$ 的哈密顿路径为 l_1，…，l_k，$l_i \in L(v)$，$1 \leqslant i \leqslant k$．由 $W(v)$ 导出子图 $G(W(v))$ 的哈密顿路径为 w_1，w_2，…，w_s，$w_j \in W(v)$，$1 \leqslant j \leqslant s$。由于 $l_k \succ v$ 且 $v \succ w_1$，故存在 G 的一条哈密顿路径 l_1，l_2，…，l_k，v，w_1，w_2，…，w_s．注意，这等于给引理 9.5 提供了另一个证明．

根据这种思想，下面给出找竞赛图的哈密顿路径的并行算法。在此之前，我们给出一个引理将对怎样选择顶点 v 起指导作用．

引理 9.6　在一个竞赛图 $G(V,E)$ 中，这里 $|V|=n$。存在这样一个顶点 $v \in V$，v 的集合 $L(v)$ 及集合 $W(v)$ 分别至少含有 $\lfloor n/4 \rfloor$ 个顶点．

证明　令 $IN=\{u \mid d_{in}(u) > d_{out}(u),\ u \in V\}$，$OUT = V - IN$，若 $|IN| \geqslant |OUT|$，由鸽巢原理，在 IN 中存在一个顶点 v，它在 $G(IN)$ 中出度不小于它的入度，即

$$d_{out}(v) \geqslant \lfloor |IN|/2 \rfloor \geqslant \lfloor n/4 \rfloor \text{ 且 } d_{in}(v) \geqslant d_{out}(v) \geqslant \lfloor n/4 \rfloor ;$$

若$|OUT| \geqslant |IN|$，我们可在 OUT 集合内取一个顶点v，其余证明与上面类似．

由引理 9.6，我们得到一个求竞赛图的哈密顿路径算法．

算法9.2　FINDING HAMILTON PATH IN TOURNAMENT

输入：竞赛图$G(V,E)$；

输出：哈密顿路径H__Path．

procedure H__path(G, $|V|$)；

begin

(1) $n \leftarrow |V|$；

(2) **if** $n=1$ **then return** the unique vertex of G **endif**；

(3) find a vertex $v \in V$ whose in__degree and out__degree in G are

both at least $\lfloor n/4 \rfloor$;

(4) **in parallel** find H_1 = H_Path($G(L(v))$, $|L(v)|$);

H_2 = H_Path($G(W(v))$, $|W(v)|$);

(5) **return** the path $((H_1 \cdot v) \cdot H_2)$

end .

定理 9.2 在 SIMD－EREW PRAM 上，计算一个竞赛图 $G(V,E)$，$|V| = n$ 的哈密顿路径，算法 9.2 需 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器。

证明 由引理 9.6 可知，算法 9.2 的第 (4) 步需执行 $O(\log n)$ 次递归调用。每次调用时，算法的第 (3) 步在最坏情况下，若采用处理器分组技术，在使用 $O(n^2/\log n)$ 处理器时，可在 $O(\log n)$ 时间内完成。故整个算法过程需 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器。

9.2.2 算法的基本原理

首先我们引入**受限的哈密顿路径** (Restricted Hamiltonian Path) 概念。一条哈密顿路径是受限的哈密顿路径，若它的其中一个端点是指定的 (注意，只有一个端点指定，而不是两个端点都指定.) . 设 $G(V,E)$ 是一个竞赛图，$v \in V$ 是这样的一个端点，若从 v 出发到所有顶点都存在一条有向路径，则称 v 为源点；若所有其它顶点都存在一条有向路径可达 v，则称 v 为终点.

定理 9.3 在一个竞赛图 $G(V,E)$ 中，如果 v 是 G 的源点 (终点) ，那么 G 有一条从 v 开始 (终止于 v) 的哈密顿路径。

证明 我们仅证明 $v \in V$ 是源点的情形. 事实上，当 v 是终点时，其证明是对称的。现在用归纳法对 n 施加证明. 当 $n = 1$ 时，命题显然成立。假定对 n 个顶点的竞赛图，命题成立。现考虑一个具有 $n+1$ 顶点的竞赛图 $G(V,E)$，令 v 是 G 的源点。运用归纳断言，我们仅需证明 $G(W(v))$ 含有 $G(V-\{v\})$ 的一个源点. 根据引理 9.5，$G(W(v))$ 含有一条哈密顿路径。比如说这条路径始于 u，则 u 就是 $G(W(v))$ 的源点。此外，根据归纳假设，$W(v)$ 中的一些顶点可达 $L(v)$ 的每个顶点，这样 u 是 $G(V-\{v\})$ 的源点.

引理 9.7 设 $G(V,E)$ 是一个竞赛图，C_1，C_2，…，C_k 是 G 的强连通分支，那么对任何 C_i 及 C_j 来讲，或者 $C_i \succ C_j$ 或者 $C_j \succ C_i (j \neq i)$，$1 \leq i,j \leq k$.

证明 根据强连通分支定义，连接 C_i 与 C_j 之间的所有有向边方向是一致的。因为 G 是一个竞赛图，这些边是存在的。

引理 9.8 设 $G(V,E)$ 是一个竞赛图，C_1、C_2，…，C_k 是它的强连通分支，根据集合间的支配关系 " $\succ$ "，一定存在一个有序序列 C_{i_1}，…，C_{i_k} 使得:

$$C_{i_1} \succ C_{i_2} \succ C_{i_3} \succ \cdots \succ C_{i_k}, \quad 1 \leq i_j \leq k, \quad 1 \leq j \leq k$$

证明 由引理 9.7 可知，对任意三个强连通分支 C_i，C_j，C_l，不妨假定 $C_i \succ C_j$，

$C_j \succ C_l$，则 $C_i \succ C_l$．我们采用反证明法证明：若 $C_l \succ C_i$，如图 9.1 所示，则从

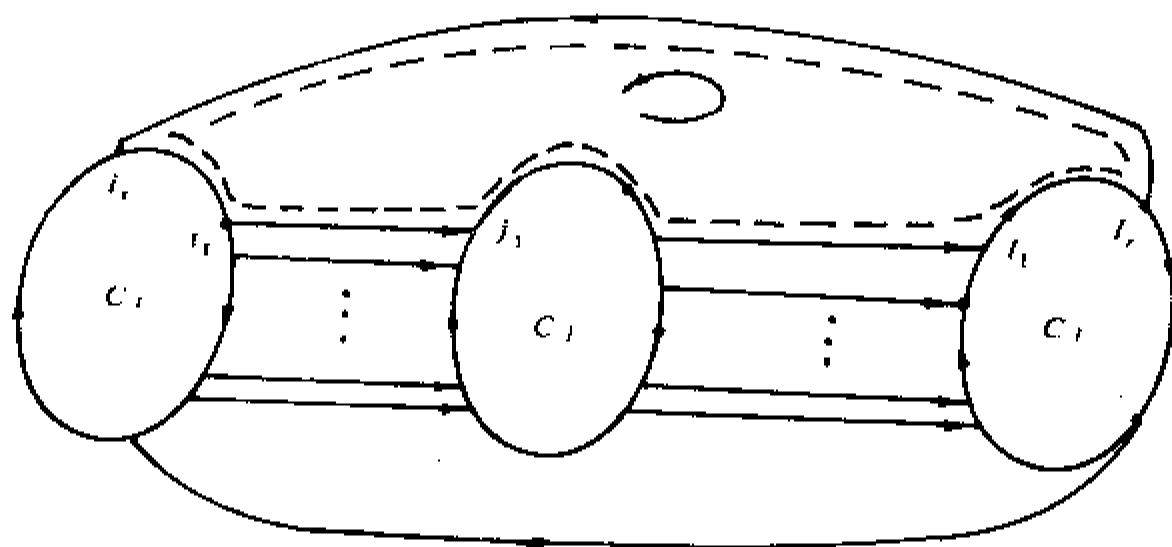


图 9.1　若 $C_i \succ C_j$，$C_j \succ C_l$，但 $C_l \succ C_i$ 情形

i_1 出发到达 j_1，再由 j_1 到 l_1，接着由 l_1 经 l_r 到达 i_t，再由 i_t 回到 i_1，形成一条回路．显然这条回路上从任一顶点到所有其它顶点都是可达的，也就是说，C_i，C_j 及 C_l 上的一些顶点也组成了一个强连通分支，这与 C_i，C_j 及 C_l 是三个不同的强连通分支矛盾．所以必有 $C_i \succ C_l$．由此类推，最终可以得出 $C_{i_1} \succ C_{i_2} \succ C_{i_3} \succ \cdots \succ C_{i_k}$．

根据上述两个引理，我们可给出定理 9.3 的另外一个证明．设 C_1，C_2，…，C_k 是 G 的强连通分支且有 $C_1 \succ C_2 \succ C_3 \succ \cdots \succ C_k$．由于 v 是 G 的源点，它必须位于 C_1 上．因 C_1 是强连通的，它含有一条哈密顿回路 H_1．令 H_2 是将 H_1 中进入 v 的那条边删去后的哈密顿路径．请注意：H_2 是 C_1 的一条始于 v 的哈密顿路径．令 H_3 是 $\{C_2, C_3, \cdots, C_k\}$ 的哈密顿路径．通过上述的构造，H_2 的最后一个顶点将支配 H_3 的第一个顶点，这样 $H_2 \cdot H_3$ 是 G 的一条从 v 开始的哈密顿路径．

定理 9.4　设竞赛图 $G(V,E)$ 是一个强连通图，那么存在一条哈密顿回路．

证明　设 G 是一个强连通竞赛图，令 $v \in V$，且 $L_1 \succ L_2 \succ L_3 \succ \cdots \succ L_q$ 是 $G(L(v))$ 强连通分支，且 $W_p \succ W_{p-1} \succ \cdots \succ W_2 \succ W_1$ 是 $G(W(v))$ 的强连通分支．由于 G 是强连通的，故必存在某条边离开 W_1，每条这样的边必指向 $L(v)$ 的一个点，根据定义，它不可能指向 W_i 的顶点 $i > 1$ 或者指向 v．令 $m' = \min\{i \mid a \succ b$ 对某个 $a \in V(W_1)$，$b \in V(l_1)\}$ 同时令 $w_1 \in V(W_1)$，$l_1 \in V(L_{m'})$ 且 $w_i \succ l_1$．对称地，必存在一条指向 L_1 顶点的边．令

$k = \min\{i \mid a \succ b$ 对某些 $a \in V(W_i)$，$b \in V(L_1)\}$，$w_2 \in V(W_k)$，$l_2 \in V(L_1)$ 且 $w_2 \succ l_2$．

为了证明图 G 的哈密顿回路的存在性，可以将图 9.2 所示的几条路径和它们的端点连

为了证明图 G 的哈密顿回路的存在性，可以将图 9.2 所示的几条路径和它们的端点连接起来，这样就构造性地证明了存在一条哈密顿回路．这几条路径分别为:

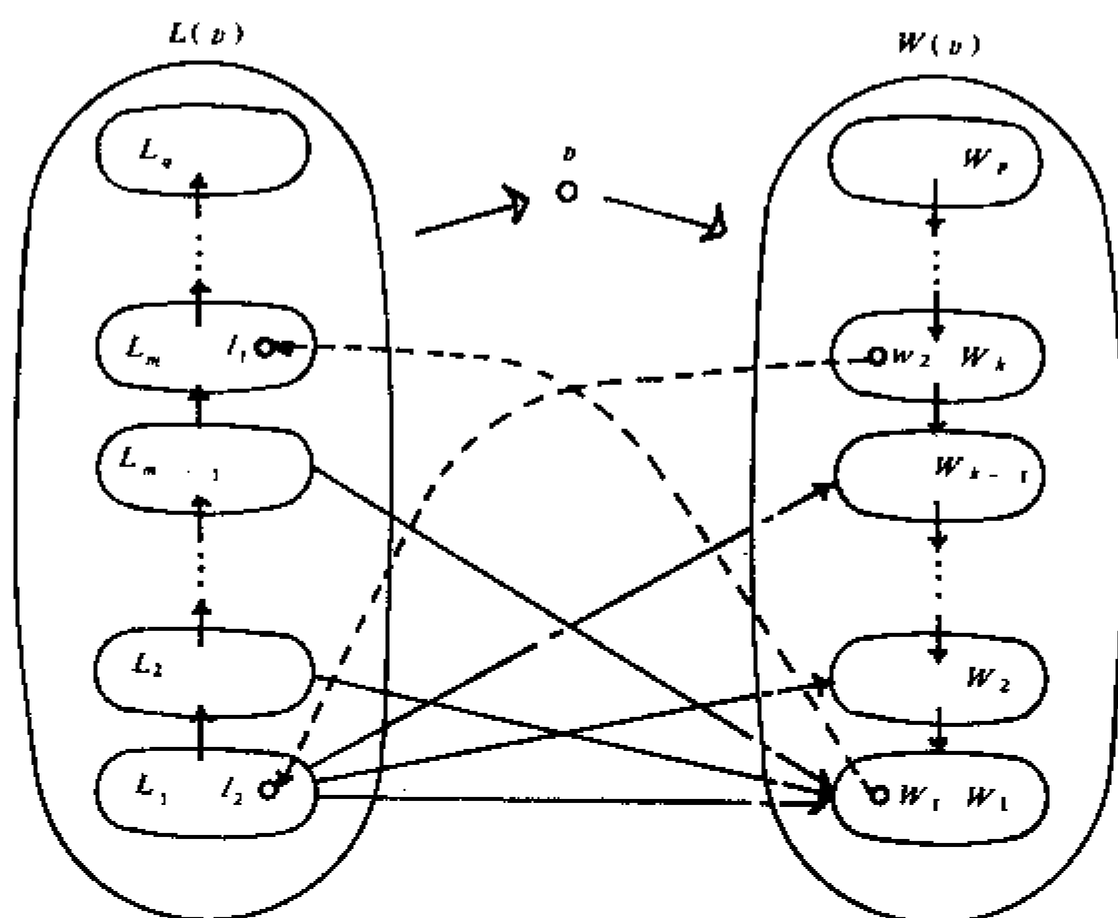


图 9.2 定理 9.4 的构造性证明．粗线表示从一连通分支到另一连通分支的有向边，虚线表示单个有向边．

(1) $G(V(W_1))$的一条终止于w_1的哈密顿路径;

(2) $G(\bigcup_{i=m'}^{q} V(L_i))$开始于$l_1$的一条哈密顿路径;

(3) 顶点v;

(4) $G(\bigcup_{j=k}^{p} V(W_j))$的一条终止于$w_2$的哈密顿路径;

(5) $G(V(L_1))$的一条始于l_2的哈密顿路径;

(6) $G(\bigcup_{i=2}^{k-1} V(w_i) \cup \bigcup_{j=2}^{m'-1} V(L_j))$的一条哈密顿路径．

我们断言上述的路径按 (1)，(2)，(3)，(4)，(5)，(6) 次序连接起来后，就形成了 G 的一条哈密顿回路．实际上，上述的路径确实都存在．受限路径 (1)，(2)，(4) 和 (5) 是定理 9.4 的结论．我们仅需验证的事实是：连接每条路径端点之间的有向边，其方向是所期望的方向。唯一不明显的情况是：从路径 (5) 到路径 (6) 之间的连接以及从路径 (6) 到路径 (1) 之间的连接。为证明这一点，我们还记得，按某种方式选取 $L_{m'}$ 及 W_k 是要使得 L_2，L_3，…，$L_{m'-1}$ 都支配 W_1，且 W_2，W_3，…，W_{k-1} 都受 L_1 支配．因此路径 (5) 的最后一个顶点必须支配路径 (6) 的第一个顶点．类似地，路径 (6) 的最后一个顶点必须支配路径 (1) 的第一个顶点．注意路径 (6) 的两个端点可能在 $L(v)$ 中，也可能在 $W(v)$ 中．

定理 9.4 的证明实际上给出了一个找强连通竞赛图的哈密顿回路的算法．这个算法首先选择一个“平庸”(Mediocre) 的顶点 $v \in V$，将一个问题划分为几个子问题．子图 (1)，(2)，(3)，(4) 及 (5)，每个至多含 $3n/4$ 个顶点．然而由强连分支 W_2，…，W_{k-1} 及 L_2，$L_{m'-1}$ 合并的子图 (6) 可能非常大，甚至它包含除 v、w_1，w_2，l_1，l_2 这些顶点之外的所有顶点，这是因为 v，w_1，w_2，l_1，l_2 保证在这个子图的外边．图 (6) 非常大的困难事实上没有关系，问题的关键是：在 (6) 中要找的哈密顿路径是不受限制的．因此我们可以利用算法 9.2 寻找这条哈密顿路径，且不必为这个子问题的规模而烦恼．因而在 n 个顶点的竞赛图上寻找哈密顿回路或受限的哈密顿路径可以分成几个类似的子问题，它们每个都不超过 $3n/4$ 个顶点．

在叙述算法之前，我们给出怎样计算竞赛图的强连通分支．对一个一般的有向图，计算其强连通分支的方法是传递闭包法，在 SIMD－CREW PRAM 上，需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器．但是，从顶点的度数出发，计算竞赛图的强连通分支有一个更简单有效的方法．

首先定义顶点 $v \in V$ 的得分 $s(v)$ (Score)，它是 v 所支配的顶点个数．一个竞赛图的得分序列是它所有顶点得分的一个列表．

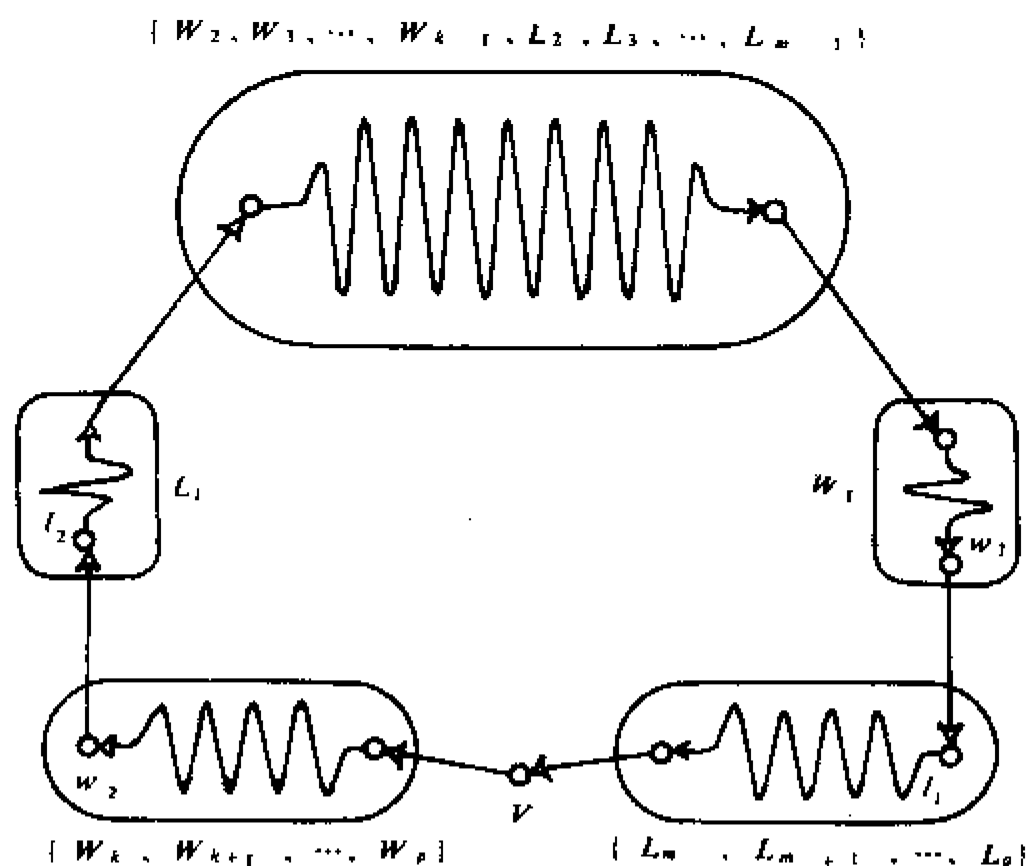


图 9.3　定理 9.4 描述的哈密顿回路（波浪线表示各个连通分支的哈密顿路径）

引理 9.9　设 $G(V,E)$ 是一个竞赛图，令 u，$v \in V$ 且 $s(u) \geqslant s(v)$，令 S_v 及 S_u 分别是含有 u 及 v 的强连通分支，那么或者 $S_u = S_v$ 或者 $S_u \succ S_v$．

证明　若 $u \succ v$，则命题显然成立．若 $v \succ u$，则根据鸽巢原理，必存在一个顶点 w，有 $u \succ w$ 且 $w \succ v$，因而命题在这种情况下也成立．

令 $V = \{v_1, \cdots, v_n\}$，这里 $s(v_1) \leqslant s(v_2) \leqslant \cdots \leqslant s(v_n)$，定理 9.4 告诉我们：对某

个 j, k，每个强连通分支的形式为 v_j, v_{j+1}, …, v_k，那么怎样决定这个强连通分支的始点及终点呢？方法很简单。如果 v_k 是一个强连 通 分 支 中 最 高 得 分顶点，那么对 $v_i \not\leqslant v_j (1 \leqslant k < j)$，有 $\sum_{i=1}^{k} s(v_i) = C_k^2$，其中 $\sum_{i=1}^{k} s(v_i)$ 是尾部在集合 $\{v_1, \cdots, v_k\}$ 中的那些有向边的数目；C_k^2 是在这个顶点集上导出的竞赛图边的数目．反之也成立。若 $\sum_{i=1}^{k_1} s(v_i) > C_k^2$，则 v_k 同 v_{k+1} 在同一强连通分支中．因此，图 G 的强连通分支计算如下：首先对每个顶点 v 计算 $s(v)$，并按非降序对 $s(v_1)$, …, $s(v_n)$ 进行排序，使得 $s(v_1) \leqslant s(v_2) \leqslant \cdots \leqslant s(v_n)$；然后计算部分和 $p_k = \sum_{i=1}^{k} s(v_i) - C_k^2$，$1 \leqslant k \leqslant n$；最后根据序列 p_1, …, p_i, …, p_n 中的零来划分强连通分支。

9.2.3 算法的形式化描述

算法 9.3 HAMILTON CYCLES ALGORITHM

proedure Restricted_Path (G, endpoint, u);

begin

(1) let n is number of vertex of G;

(2) **if** $n = 1$ **then return** the unique vertex of G **endif**;

(3) find strongly connected componets $C_1 \succ C_2 \succ, \cdots, \succ C_k$ of G;

(4) **if** endpoint = "start" **then**

(4.1) **in parallel** find $H_1 \leftarrow \text{CYCLE}(C_1)$

$$H_2 \leftarrow \text{H} - \text{Path} \ (\bigcup_{i=2}^{k} C_k, |\bigcup_{i=2}^{k} V(C_k)|);$$

(4.2) $H_1 \leftarrow H_1 - \{\text{unique directed edge into } u\}$

endif;

(5) **if** endpoint = end **then**

(5.1) **in parellel** find $H_1 \leftarrow \text{H_Path}(\bigcup_{i=1}^{k-1} C_k, |\bigcup_{i=1}^{k-1} V(C_k)|)$;

$$H_2 \leftarrow \text{CYCLE}(C_k);$$

(5.2) $H_2 \leftarrow H_2 - \{\text{unique directed edge and of } u\}$

endif;

(6) **return** the path $H_1 \cdot H_2$

end;

```
procedure CYCLE(G);            /*主过程*/
begin
(1) let n is number of vertex of G;
(2) if n = 1 then return the unique vertex of G endif;
(3) find a vertex, v∈V(G), whose in_degree and out_degree in G are both
      at least ⌊n/4⌋;
(4) find strongly connected components L_1 ≻ L_2 ≻ … ≻ L_q of G(L(v)) and
      W_p ≻ W_{p-1} ≻ … ≻ W_1 of G(W(v));
(5) in parallel find
      m' ← min{i | a ≻ b for some a∈V(W_1), b∈V(L_i)};
      k ← min{i | a ≻ b for some a∈V(W_i), b∈V(L_1)};
      the results are w_1∈V(W_1), l_1∈V(L_m'), w_2∈V(W_k), l_2∈V(L_1),
      and w_1 ≻ l_1, w_2 ≻ l_2;
(6) in parallel find
      H_1 ← Restricted_Path(W_1, "end", w_1);
      H_2 ← Restricted_Path(∪_{i=1}^{m'} L_i, "start", l_1);
      H_3 ← Restricted_Path(∪_{i=k}^{p} W_i, "end", w_2);
      H_4 ← Restricted_Path(L_1, "start", l_2);
      H_5 ← H_Path(∪_{i=2}^{k-1} W_i ∪ ∪_{j=2}^{m'-1} L_j, |∪_{i=2}^{k-1} V(W_i)| + |∪_{j=2}^{m'-1} V(L_j)|;
(7) return the cycle v·H_3·H_4·H_5·H_1·H_2·v
end.
```

9.2.4 算法的复杂性分析

在给出算法的复杂性之前，我们首先给出计算竞赛图的强连通分支的复杂性。

引理 9.10 在 SIMD－EREW PRAM 上，求一竞赛图 $G(V,E)$，$|V|=n$，的所有强连通分支，需 $O(\log n)$ 时间、$O(n^2/\log n)$ 处理器。

证明 首先，计算每个顶点 $v\in V$ 的得分 $s(v)$，需 $O(\log n)$ 时间、$O(n^2/\log n)$ 处理器，对竞赛图得分序列进行排序，需 $O(\log n)$ 时间、$O(n)$ 处理器[6]．然后，对得分序列计算部分和，可在 $O(\log n)$ 时间内使用 $O(n/\log n)$ 处理器完成[10]。故求 G 的强连通分支需 $O(\log n)$ 时间、$O(n^2/\log n)$ 处理器。

定理 9.5 在 SIMD－EREW PRAM 上，找一竞赛图 $G(V,E)$，$|V|=n$ 的受限哈密顿路

径或哈密顿回路，需 $O(\log^3 n)$ 时间、$O(n^2/\log n)$ 处理器。

证明 找一竞赛图的受限哈密顿路径或哈密顿回路，是由过程 Restricted_Path 及过程 CYCLE 相互调用获得的。因每次 v 的选择使其入度和出度均大于等于 $\lfloor n/4 \rfloor$，故得到子问题的最大规模为 $\lceil 3n/4 \rceil$，所以两个过程相互调用要进行 $O(\log n)$ 次。而由定理 9.2 可知，找 n 个顶点的哈密顿路径需 $O(\log^2 n)$ 时间、$O(n^2/\log n)$ 处理器；所以算法的计算时间 $T(n)$ 为:

$$T(n)=c'\sum_{k=1}^{c\log n}\log^2((\frac{3}{4})^k n)=c'\sum_{k=1}^{c\log n}(k\log\frac{3}{4}+\log n)^2=O(\log^3 n)$$

其中c，c'为大于零的常数。处理器数仍为$O(n^2/\log n)$。

9.3 小　　结

本章主要介绍了在欧拉图中找欧拉回路以及竞赛图中找哈密顿回路的并行算法。本章讨论的欧拉回路的并行算法将在后面一些章节中作为子过程被调用。

值得指出的是：虽然 Awerbuch 等人的算法基本思想同本章介绍的算法 9.1 是一致的，但算法 9.1 看起来似乎更容易懂。尤其应该指出的是：Tarjan 等人对树边赋予方向相反的两条边后变成的欧拉图，找这个欧拉图的欧拉回路（路径）非常容易，他们利用树上欧拉回路使得树上许多操作，如求两个顶点的最低公共祖先，计算每个顶点离根的深度等问题的时间复杂性变为 $O(\log n)$，而仅使用 $O(n)$ 处理器，较一般计算方法在使用处理器方面降低了 $O(n)$ 因子[4,5]。

对于一般的图，并行地找它的哈密顿路径或回路，目前研究还不是太多，因为这个问题的串行算法目前仍是指数复杂性的[11]。但对一些特殊的图，其哈密顿路径或回路已有人做了不少工作，如 Soroker 等人曾独立地给出找竞赛图的哈密顿路径的并行算法[9]。Bertossi 等人在 SIMD-CREW PRAM 上，给出了找真区间(Proper Interval Graph)的哈密顿路径（回路）算法。他们的算法需 $O(\log n)$时间、$O(n^2)$处理器[13]。1989 年，作者将这个复杂性改进到 $O(\log n)$时间、$O(n)$处理器[14]。最近，基于 SIMD-CREW PRAM，Dahlhans 等人对稠密图（无向图的最小度数为 $n/2$）给出了一个 $O(\log^5 n)$时间、$O(n^4)$处理器的找哈密顿回路的算法[15]。

参 考 文 献

[1] Atallah M, Vishkin V. Finding Euler Tours in Parallel, *J. Comput and System Sci.*, **29**, 1984, 330–337

[2] Awerbuch B, ISraeli A, Shiloach Y. Finding Euler Circuits in Logorithmic Parallel Time, *In Proc. 6th ACM Symp. On Theory of Computing*, 1984, 249–257

[3] Shiloach Y, Vishkin U. An O(log*n*) Parallel Algorithm, *J. Algorithms*, **3**, 1982, 57–67

[4] Tarjan R E, Vishkin U. An Efficient Parallel Biconnectivity Algorithm, *SIAM J. Comput.*, **14**, 1985, 862–874

[5] Tarjan R E, Vishkin U. Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time, *24th Annu. Symp. on FOCS*, 1984, 12–20

[6] Cole R. Parallel Merge Sort, *SIAM J. Comput.*, 17(4), 1988, 770–785

[7] Eckstein D M. Simultaneous Memory Access, TR–79–6, Computer Science Department, Iowa State Univ., Ames., Iowa,1979

[8] Vishkin U. Implementation of Simultaneous Memory Address Access in Models That Forbit It. *J. Algorithms*, **4**, 1983, 45–50

[9] Soroker D. Fast Parallel Algorithms for Finding Hamiltonian Path and Cycles in a Tournament, *J. Algorithms*, **9**, 1988, 276–286

[10] Ladner R E, Fischer M J. Parallel Prefix Computation, *J.ACM*, **27**(4), 1980, 831–838

[11] Garey M, Johnson D. *Computers and Intractability: A Guide to the Theory of NP–Completeness*, W.H. Freeman, 1979

[12] Naor J. Two Parallel Agorithms in Graph Theory, Tech. Rept, CS–86–6, Hebrew Univ., 1986

[13] Bertossi A A, Bonuccelli M.A., Some Parallel Algorithms on Interval Graphs, *Discrete Applied Math.*, **16**, 1987, 101–111

[14] 唐策善，梁维发．关于区间图的一些有效并行算法，高校应用数学学报，4（4），1989，534–539

[15] Dahlhans E, Karpinski M. Parallel Construction of Perfect Matchings and Hamiltonian Cycles on Dense Graphs, *Theoretical Comput. Sci.*, **61**, 1988, 121–136

第十章　流图的并行算法

流图在工程上有广泛的应用背景。AOE 网(Activity−On−Edge Network)，又称分析活动网）问题在系统工程、网络分析等领域中经常出现，求一个网络的最大流最小割问题是许多网络优化设计中常常遇到的问题。本章我们介绍这两个问题的并行算法。

10.1　共享存贮模型上的 AOE 网问题的并行算法

AOE 网是一个无有向回路的有向图 $G(V,E)$。图中每个顶点 $i\in V$ 对应一个事件，每条有向边 $<i,j>\in E$ 对应网上的一个活动，且每条有向边赋给一个正的权值，它表示该边上的活动成本。不失一般性，假定 $V=\{1, 2, \cdots, n\}$，s 代表源点（起始点），t 代表终点，$s\in V$，$t\in V$。

AOE 网问题的并行算法涉及三个子问题：第一，已知一个有向图，其边赋给正的权值 w 后，该有向图是否是一个 AOE 网；第二，若一个有向图 $G(V,E)$ 是一个 AOE 网，试给出每个事件的执行次序，即求 AOE 网的拓扑排序；第三，试找出 AOE 网的关键路径。在下面各节，我们将分别介绍这些问题在 SIMD − CRCW PRAM 上的并行算法。

10.1.1　AOE网的存在性测试算法

检查一个有向图 $G(V,E)$ 是不是 AOE 网，其必要条件是：检测 G 是否含有一个有向回路，这是因为 AOE 网不能存在有向回路。本节介绍的检测算法是 Chaudhuri 等人建议的[1]。

1. 算法的基本原理

检测一个有向图 $G(V,E)$ 是否含有向回路的一个直观方法是：首先计算 G 的可达性矩阵 R。R 的元素 $R(i,j)$ 定义为：若顶点 i 到顶点 j 之间存在一条有向路径，则 $R(i,j)=1$；否则，$R(i,j)=0$。其次判断是否存在一条边 $<i,j>\in E$，使得 $R(i,j)\wedge A(j,i)\neq 0$。这里 A 是 G 的邻接矩阵。若 $R(i,j)\wedge A(j,i)\neq 0$，则意味着 $R(i,j)\neq 0$(即是 1)，同时 $A(j,i)\neq 0$(即是 1)。也就是说，从 i 到 j 存在一条有向路径，从 j 到 i 存在一条有向边。这条有向路径及边 $<j, i>$ 形成 G 中的一条有向回路。因此，一个有向图 $G(V,E)$ 是AOE 网当且仅当对任意一条边 $<i,j>\in E$ 均有 $R(j,i)\wedge A(i,j)=0$。R 的计算可通过计算 G 的邻接矩阵 A 获得。即

$$R=(A+I)^n=(\cdots(((A+I)^2)^2)^2\cdots)^2 \text{①}$$

其中 I 是 n 阶单位矩阵。

① 含有 ⌈log$n$⌉重括号，即有 ⌈log$n$⌉个 2

2. 算法的形式化描述

```
算法10.1  TEST ALGORITHM
    输入: 图G的邻接矩阵A;
    输出: 图G是不是包含有向回路, 用"Yes"和"No"回答.
    begin
    (1) R←A+I;       /* A的自反闭包R */
    (2) for ⌈logn⌉ iterations do        /* 计算A的自反传递闭包 */
                R←R*R                   /*   "*"是合取    */
        endfor;
    (3) for each i, j: 1≤i,j≤n pardo
            if R(i,j)∧A(j,i)≠0    then Temp(i,j)←1   else Temp(i,j)←0     endif
            /* 满足条件则存在有向回路 */
        endfor;
    (4) for each i, j: 1≤i, j≤n pardo
            B(i)←max{Temp(i,j) | 1≤j≤n}
        endfor;
    (5) for each i: 1≤i≤n pardo
            x←max{B(i) | 1≤i≤n}
        endfor;
    (6) if x=1  then return("Yes")    else return("No")  endif
            /* 若存在有向回路表示G不是AOE网 */
    end.
```

定理 10.1 在 SIMD－CRCW PRAM 上，测试一个有向图 $G(V,E)$，$|V|=n$ 是否包含有向回路，算法 10.1 需要 $O(\log n)$ 时间、$O(n^3)$ 处理器。

证明 由算法 10.1 可知，第 (1) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (2) 步涉及到两个布尔矩阵的逻辑乘问题。两个布尔矩阵的逻辑乘，在 SIMD－CRCW PRAM 上，需 $O(1)$ 时间、$O(n^3)$ 处理器。因为允许同时向某一单元"写"，使得计算 $\bigvee_{k=1,2,\cdots,n}(a_{ik}\wedge b_{kj})$，可使用 $O(n)$ 处理器在 $O(1)$ 时间内完成。故第 (2) 步需 $O(\log n)$ 时间、$O(n^3)$ 处理器；第 (3) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (4) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (5) 步需 $O(\log n)$ 时间、$O(n)$ 处理器；第 (6) 步需 $O(1)$ 时间及处理器。所以，算法 10.1 需 $O(\log n)$ 时间、$O(n^3)$ 处理器。

10.1.2 AOE网的拓扑排序算法

一个 AOE 网 $G(V,E)$ 的拓扑排序是指对图的顶点（事件）进行排序，且排序后每个

顶点 $i \in V$ 有一个唯一的拓扑序号 $T(i)$，若顶点 i 是顶点 j 的前趋顶点，则 i 应排在 j 的前面，即 $T(i) < T(j)$

1. 算法的基本原理

并行拓扑排序的基本思想是：首先计算每个顶点 $i \in V$ 可达的顶点个数，然后将顶点分放到 n 个桶中。放入桶中的方法是：若从顶点 i 出发可以到达的顶点数目为 j，则把顶点 i 放入第 j 个桶中．我们用二维数组 $B(1{:}n; 1{:}n)$ 来表示 n 个桶，数组 B 的每一列 j 代表一个桶 $(1 \leqslant j \leqslant n)$。每个桶的 n 个元素中每一个代表一个唯一的单元。因此桶 j 的第 i 个单元是 $B(i,j)$．开始时，n 个桶的每个单元赋初值为 0；将一个顶点放入某个桶的一个单元中，只需将此桶的这个单元赋值 1．最后，并行地修改每个桶的单元的值，使得修改后的值是所有前面各桶单元值的和、再加上本桶所有前面单元（包括计算单元本身）的值之和。即对给定的 i，j 来讲，$B(i,j)$ 为：

$$B(i,j) = \sum_{q=1}^{j-1} \sum_{p=1}^{n} B(p,q) + \sum_{l=1}^{i} B(l,j), \quad 1 \leqslant i,j \leqslant n$$

然后由 B 计算每个顶点 $i \in V$ 的拓扑序号 $T(i)$。

2. 算法的形式化描述

算法10.2　TOPOLOGICAL SORTING

```
输入: G的邻接矩阵A;
输出: 每个顶点 i∈V 的拓扑序号 T(i).
begin
(1)  R ← A + I;
(2)  for ⌈logn⌉ iterations do        /* 计算G可达性矩阵R */
         R ← R * R
     endfor;
(3)  for each i, j: 1 ≤ i,j ≤ n pardo       /* 计算每个顶点可达的顶点个数 */
         C(i) ← Σ_{j=1}^{n} R(i,j) ;   X(i) ← 0 ;     B(i,j) ← 0
     endfor;
(4)  for each i,j : 1 ≤ i, j ≤ n pardo     /* 对每个桶赋初值 */
         if C(i) = j   then B(i,j) ← 1    endif
     endfor;
(5)  for each i,j : 1 ≤ i,j ≤ n pardo     /* 计算每个桶的部分和 */
         for each p : 1 ≤ p ≤ i pardo
             B(i,j) ← Σ_{p=1}^{i} B(p,j)
         endfor
     endfor;
(6)  for each j : 1 ≤ j ≤ n pardo       /* 计算编号小于等于j的桶所含数据总数 */
         for each q: 1 ≤ q ≤ j pardo
```

$$X(j) \leftarrow \sum_{q=1}^{j} B(n,q)$$

endfor

endfor ;

(7) **for** each i,j : $1 \leqslant i,j \leqslant n$ **pardo** / * 计算部分和 * /

$B(i,j) \leftarrow X(j-1) + B(i,j)$

endfor ;

(8) **for** each i: $1 \leqslant i \leqslant n$ **pardo** / * 计算顶点的拓扑序号 * /

PSUM(i) ← $B(i,C(i))$; $T(i) \leftarrow n -$ PSUM(i) + 1

endfor

end .

下面我们证明算法 10.2 确实能正确地给出 AOE 网的拓扑排序。在一个 AOE 网中，若顶点 i 是顶点 j 的前趋顶点，则顶点 j 可达的顶点 i 也可达。这是因为从 i 可达 j；反之 i 可达的顶点 j 未必能到达。即 $C(i) > C(j)$，这意味着 PSUM(i) > PSUM(j)，故 $T(i) < T(j)$。这同拓扑排序的定义一致。

定理 10.2 在 SIMD－CRCW PRAM 上，对一个 AOE 网 $G(V,E)$，$|V| = n$ 的所有顶点进行拓扑排序，算法 10.2 需 $O(\log n)$ 时间、$O(n^3)$ 处理器。

证明 从算法 10.2 可以看出，算法的第 (1)～(2) 步同算法 10.1 第 (1)～(2) 步相同，需 $O(\log n)$ 时间、$O(n^3)$ 处理器。第 (3) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (4) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (5)～(6) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (7) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (8) 步需 $O(1)$ 时间、$O(n)$ 处理器。故整个算法需 $O(\log n)$ 时间、$O(n^3)$ 处理器。

图 10.1 给出了一个 AOE 网的例子，用算法 10.2 计算它的拓扑排序的结果，列在表 10.1 中。

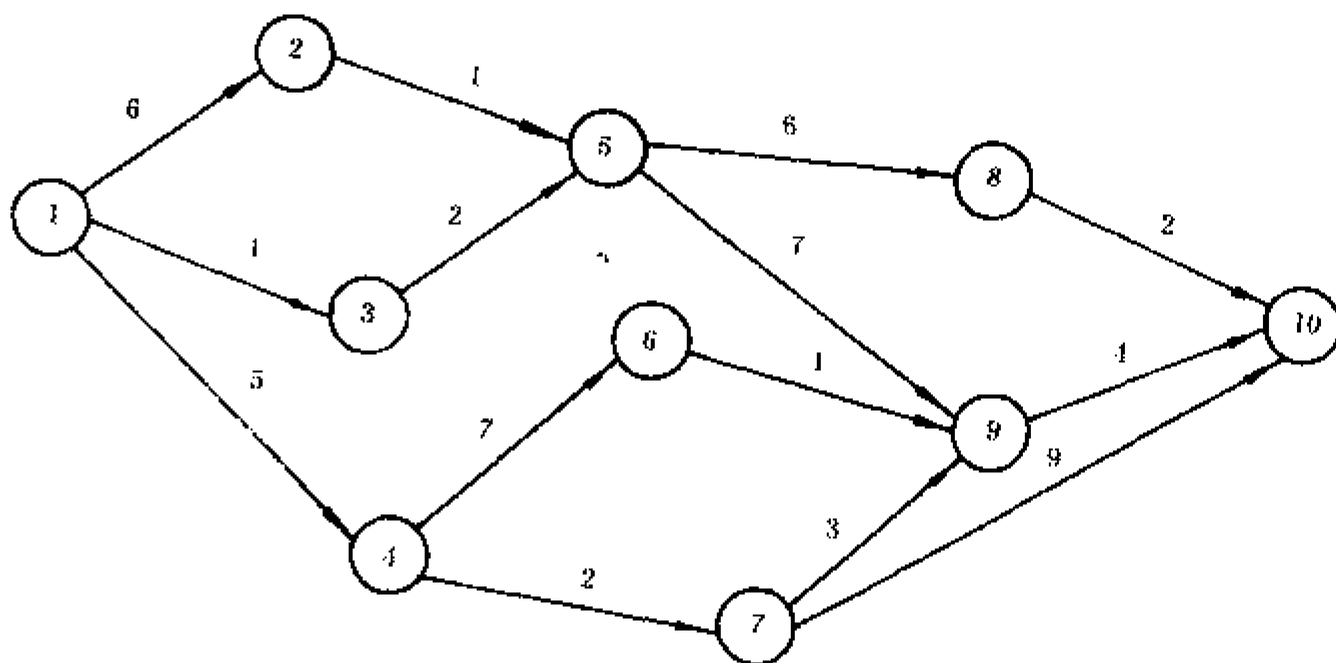


图 10.1 AOE 网示例

表 10.1 算法 10.2 对图 10.1 的执行结果

顶点或事件(i)	$C(i)$	PSUM(i)	$T(i)$
1	10	10	1
2	5	7	4
3	5	8	3
4	5	9	2
5	4	6	5
6	3	4	7
7	3	5	6
8	2	2	9
9	2	3	8
10	1	1	10

10.1.3 AOE网的关键路径算法

在一个工程项目中，找出反应工程进展的关键环节，对整个工程能否按期完成至关重要。这就是在 AOE 网中找出一条从源点 s 出发至终点 t 的关键路径。

1. **求最大加权路径的权值**

在介绍关键路径算法之前，我们首先给出计算有向加权图的所有顶点对之间的最大加权路径权值的算法。

设 $G(V,E)$ 是一个有向加权图，d 是图 G 的直径。一棵根为 $x\in V$ 的 树 T_x^k 定义为：T_x^k 是由从 x 出发、经过图中非有向回路的有向路径到达的所有顶点组成，且从 x 到这些顶点的路径长度不超过 2^k，$0\leqslant k\leqslant \lceil \log d\rceil$。在树 T_x^k 中，若存在这样一对顶点 y 及 z，$<y,z>\in E$，但 $<y,z>$ 不是 T_x^k 的边。若下列一个条件满足，则说树 T_x^k 保持最大加权路径性质：

(1) 在 T_x^k 中从 x 到 y 的路径由 2^k 条边组成，每条边权值用 $W(y|T_x^0)$ 表示，在 T_x^k 中从 x 到 y 路径上所有边的权值和记为 $W(y|T_x^k)$；

(2) $W(y|T_x^k)+W(z|T_y^0)\leqslant W(z|T_x^k)$。

在树 T_x^k 中，若边 $<y,z>$ 属于 T_x^k 的边，则顶点 z 的父顶点 $F(z|T_x^k)=y$ 。约定树根 x 的 $F(x|T_x^k)=x$。

找一个有向加权图的所有最大路径树算法的基本思想是：首先假定算法的输入是：对每个 $x\in V$，设输入为 T_x^0，若 $<y,x>\in E$，则输入权值是 $W(y\mid T_x^0)\leftarrow W(y,x)$；否则 $W(y|T_x^0)\leftarrow -\infty$。算法开始时，对每个 $x\in V$，将所有满足 $y\in T_x^0$ 的树 T_y^0 同 T_x^0 合并，产生一个新的树 T_x^1。然后再类似地合并下去，合并过程执行 $\lceil \log d\rceil$ 次循环后，对每个 $x\in V$ 得到一棵树 $T_x^{\lceil \log d\rceil}$，它是根在 x 的一棵最大加权路径树。在 每 次 循 环 时，

树 T_x^k 是由树 T_x^{k-1} 和所有树 T_y^{k-1} $(y \in T_x^{k-1})$ 合并得出的，且保持最大加权路径性质 $(0 \leq k \leq \lceil \log d \rceil)$.

2. **算法的形式化描述**

算法10.3 FINDING MAXIMUM WEIGHTED PATH

输入：树 T_x^0，对每个 $x \in V$，用 $F(y|T_x^0)$ 表示树 T_x^0，$\forall y \in V$;

输出：对每个 $x \in V$，输出根在 x 的最大加权路径树，用 $F(y \mid T_x^{\lceil \log d \rceil})$ 表示 $T_x^{\lceil \log d \rceil}$，

$y = 1, 2, \cdots, n$，且 $y \in T_x^{\lceil \log d \rceil}$.

```
procedure Max_path(G);
  begin
  (1) k ← 0;
  (2) while k ≤ ⌈logd⌉ do        /* 计算最大路径矩阵M_x^k */
  (3)     for each x, y, z: 1 ≤ x, y, z ≤ n pardo
              if (z = x)∧(F(y|T_x^k) ≠ 0)
                  then M_x^k(y,z) ← W(y|T_x^k)
                  else if (z∈T_x^k)∧(F(y|T_z^k) ≠ 0)
                      then M_x^k(y,z) ← W(z|T_x^k) + W(y|T_x^k)
                      else M_z^k(y,z) ← −∞
                  endif
              endif
          endfor;
  (4)     for each x, y, z: 1 ≤ x, y, z ≤ n pardo   /* M_x^k矩阵每行求出最大值 */
              M_x^k(y,y_m) ← max {M_x^k(y,z) | z = 1, 2, …, n}
          endfor;
  (5)     k ← k + 1;
  (6)     for each x,y: 1 ≤ x,y ≤ n pardo   /* 调整最大路径树的父结点 */
              F(y |T_x^k) ← F(y |T_{y_m}^{k-1})
          endfor;
  (7)     for each x,y: 1 ≤ x,y ≤ n pardo   /* 修改最大路径边上的权值 */
              W(y|T_x^k) ← W(y_m|T_x^{k-1}) + W(y|T_{y_m}^{k-1})
          endfor
      endwhile
   end.
```

3. 算法的正确性证明

在分析算法 10.3 复杂性之前，先对它的正确性给予证明。

引理 10.1 对任意的 $x\in V$，算法 10.3 经过 k 次连续循环后得出的树 T_x^k，是由从 x 出发可达的顶点组成的，且从 x 到这些顶点的非有向回路路径长度 $\leqslant 2^k$，$0\leqslant k\leqslant \lceil \log d\rceil$。

证明 对任意 $x\in V$，用归纳法对 k 施加证明。当 $k=0$ 时，引理显然成立。因为根据 T_x^0 定义，它仅由 $(x,y)\in E$ 的顶点 y 组成。假定在 $i=k$ 时，每个 T_x^i 使得引理 10.1 成立，$x\in V$。现证明 $i=k+1$ 时引理 10.1 成立。考虑一个顶点 $y\notin T_x^{i+1}$，但在图 G 中存在一条长度小于等于 2^{i+1} 的非有向回路。由算法 10.3 的第 (3)～(6) 步可知：若 $y\notin T_x^{i+1}$，则意味着存在这样的顶点 u，使得 $u\in T_x^i$ 且 $y\in T_u^i$。由归纳假设，T_u^i 含有图 G 的那种顶点集合，即从 u 出发长度小于等于 2^i 的非有向回路的可达顶点集合。因此，对任意 $u\in T_x^i$ 来讲，当 $y\notin T_x^i$ 且 $y\notin T_u^i$，则在图 G 中从 x 到 y 的每条非有向回路长度大于 $2^i+2^i=2^{i+1}$，这与关于 y 的归纳假设相矛盾。故对每个顶点 $y\in V$，若从 x 出发可达它的非有向回路路径长度小于等于 2^{i+1}，则 $y\in T_x^{i+1}$。

引理 10.2 对任意一个 $x\in V$，经算法 10.3 的第 (3)～(6) 步连续循环后的树 T_x^k，将不包括 $G(V, E)$ 的那样的顶点 y，使得 G 中从 x 出发到达 y 的非有向回路路径长度大于 2^k，$0\leqslant k\leqslant \lceil \log d\rceil$。

证明 类似引理 10.1 的证明。

引理 10.3 在算法 10.3 的第 (3)～(6) 步的第 k 次循环结束时，对任意的 $x\in V$，由 T_x^{k-1} 以及所有 $T_y^{k-1}(y\in T_x^{k-1})$，形成的合并树 T_x^i，保持最大加权路径性质。

证明 归纳基础。$k=0$ 时，由 T_x^0 定义，T_x^0 保持最大加权路径性质，$x\in V$。归纳假设。设对一个整数 $k=i$ 来讲，每棵树 T_x^i 保持最大加权路径性质。现考虑 $k=i+1$ 的树 T_x^{i+1}，它是由 T_x^i 及所有 $T_u^i(u\in T_x^i)$ 的树合并后形成的。对每个 $y\in T_u^i$，$u\in T_x^i$ 的顶点 y。我们在 T_x^{i+1} 选择 y 的父顶点 $z=F(y|T_x^{i+1})$，使得在图 G 中不再存在另一个顶点 $u(u\neq z)$，满足：

$$W(u\mid T_x^{i+1})+W(y\mid T_u^o)>W(y_m\mid T_x^i)+W(z\mid T_{y_m}^i)+W(y\mid T_Z^0)$$

由算法 10.3 的第 (4)～(6) 可知，这样的选择是可行的。根据归纳假定，每棵树 T_x^i 保持最大加权路径，$x\in V$。因为在图中任意一条从 x 到 y 的路径，若 T_x^i 及 $T_{y_m}^i$ 都保持这条路径的最大加权路径性质，则最大加权路径的权值必须大于等于 $W(y_m\mid T_x^i)+W(z\mid T_{y_m}^i)+W(y\mid T_x^o)$。然而我们可以选择 $F(y\mid T_x^{i+1})$，使得

$$W(y \mid T_x^{i+1}) = W(y_m \mid T_x^i) + W(z \mid T_{y_m}^i) + W(y \mid T_z^o)$$

因此，T_x^{i+1}保持最大加权路径性质。

由上述诸引理可知，对每个顶点 $x \in V$，树 $T_x^{\lceil \log d \rceil}$ 是图 G 的以 x 为根的最大加权路径树。

定理 10.3 在 SIMD－CRCW PRAM 上，对一个非有向回路的有向加权图 $G(V,E)$，$|V| = n$，计算所有顶点 $x \in V$ 的最大加权路径树 $T_x^{\lceil \log d \rceil}$，算法 10.3 需 $O(\log d \log\log n)$ 时间、$O(n^3)$ 处理器，这里 d 是图 G 的直径。

证明 通过对算法 10.3 复杂性分析来证明定理。第 (1) 步需 $O(1)$ 时间及处理器；第 (3) 步需 $O(1)$ 时间、$O(n^3)$ 处理器；第 (4) 步需求 n 个元素最大值运算，运用 Shiloach 等人的算法[2]，求 n 个元素最大值需 $O(\log\log n)$ 时间、$O(n)$ 处理器，故第 (4) 步需 $O(\log\log n)$ 时间、$O(n^3)$ 处理器；第 (5)～(6) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；而 while 循环需执

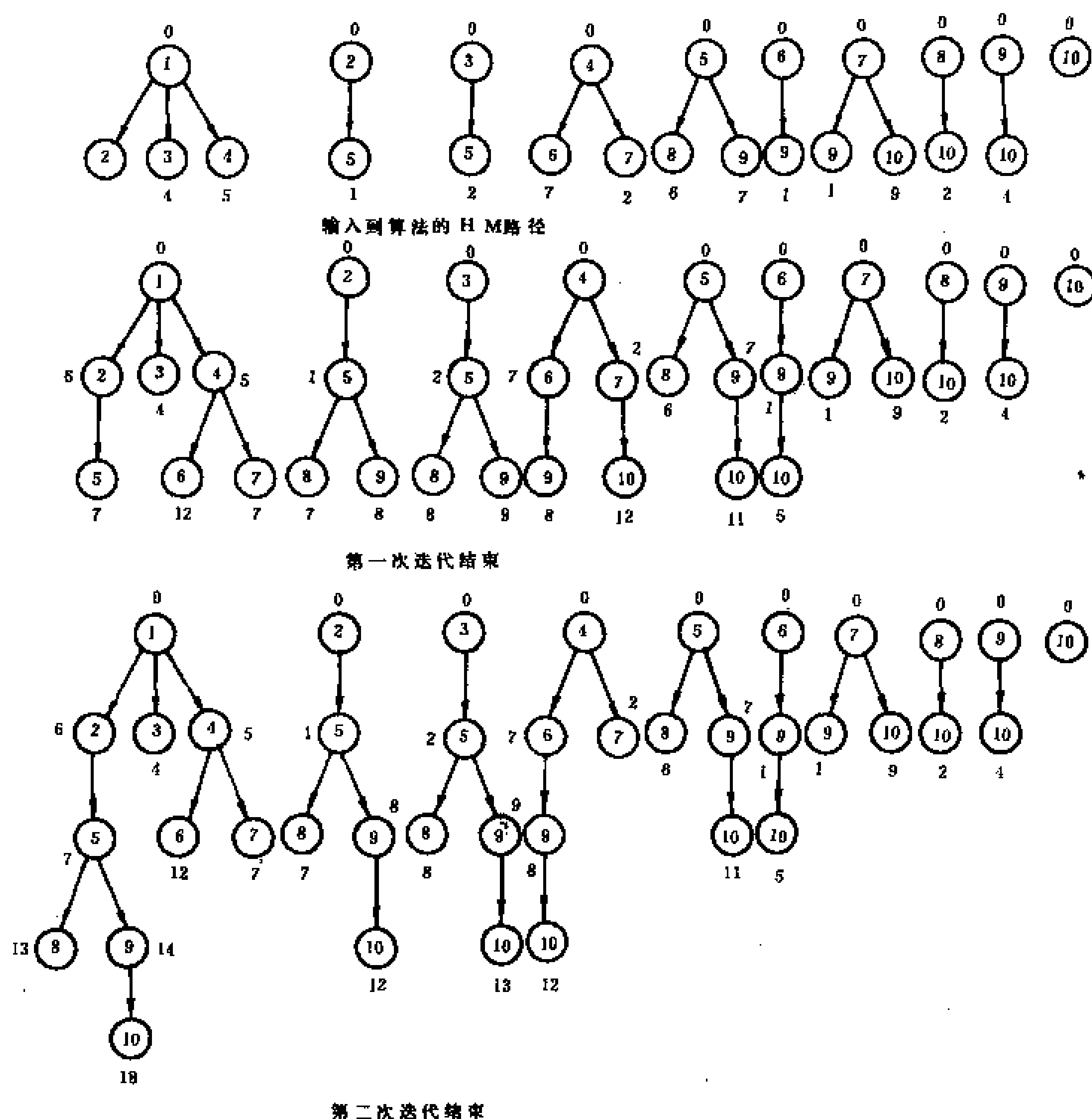


图 10.2 算法 10.3 对图 10.1 的执行结果

行 $\lceil \log d \rceil$ 次．故整个算法10.3需$O(\log d \log\log n)$时间、$O(n^3)$处理器．

图 10.2 给出了图 10.1 在执行算法 10.3 的前两次循环后的各个最大加权路径树．

4. **求AOE网的关键路径**

设每个活动 $<i,j> \in E$ 的最早开始时间为 EST(i,j)，最迟开始时间为 LST(i,j)，松弛时间 (Slack Time) 为 SLACK(i,j)．显然，

$$\text{SLACK}(i,j) = \text{LST}(i,j) - \text{EST}(i,j)$$

一个活动 $<i,j> \in E$ 是关键活动当且仅当 SLACK$(i,j) = 0$．因而一条从源点 s 到终点 t 的路径 P 是关键路径当且仅当对 P 上每一条边 $<x,y>$ 均有 SLACK$(x,y) = 0$．这一性质给出了计算关键路径的算法．下面我们给出形式化描述的算法．

算法 10.4 FINDING CRITICAL PATH ON AOE NETWORKS

输入：G的邻接矩阵 A 及加权邻接矩阵 W；

输出：松弛矩阵SLACK，SLACK$(i,j) = 0$表示边 $<i,j>$ 是一个关键活动．

```
procedure Critical_Activities(G);
  begin
  (1)  call Max_path(G);      /* 计算G的最大路径树 */
  (2)  for each i, j : 1 ≤ i,j ≤ n pardo
            /* 计算每个活动 <i,j> 的最早开始时间及最迟开始时间 */
         LST(i,j) ← W(t | T_s^⌈logd⌉) − W(t | T_j^⌈logd⌉) − W(j | T_i^0);
         EST(i,j) ← W(i | T_s^⌈logd⌉)
       endfor;
  (3)  for each i,j: 1 ≤ i,j ≤ n pardo    /* 计算每个活动 <i,j> 的延迟时间 */
          if <i,j> ∈ E then SLACK(i,j) ← LST(i,j) − EST(i,j)
                          else SLACK(i,j) ← −1      /* 特殊标记 */
          endif
       endfor
  end.
```

上述过程执行结果，在某条从 s 至 t 的路径上所有的边 $<i,j> \in E$ 均有 SLACK$(i,j) = 0$，则这条路径是 AOE 网的关键路径．

定理 10.4 在 SIMD－CRCW PRAM 上，计算一个 AOE 网 $G(V,E)$，$|V| = n$ 的关键路径，算法 10.4 需 $O(\log d \log\log n)$ 时间、$O(n^3)$ 处理器，这里 d 是图的直径。

证明 在算法 10.4 中，根据定理 10.3，第 (1) 步需 $O(\log d \log\log n)$ 时间、$O(n^3)$ 处理器；第 (2)～(3) 步分别需 $O(1)$ 时间、$O(n^2)$ 处理器。因此，整个算法需 $O(\log d \log\log n)$ 时间、$O(n^3)$ 处理器。

对图 10.1 的 AOE 网，每个活动 $<i,j> \in E$ 的最早开始时间 EST(i,j)，最迟开始时间 LST(i,j) 和松弛时间 SLACK(i,j) 列出在表 10.2 中。

表 10.2 图 10.1 的每个活动的最早开始时间、最迟开始时间以及延迟时间

活动 <i,j>	EST(i,j)	LST(i,j)	SLACK(i,j)
<1,2>	0	0	0
<1,3>	0	1	1
<1,4>	0	1	1
<2,5>	6	6	0
<3,5>	4	5	1
<4,6>	5	6	1
<4,7>	5	7	2
<5,8>	7	10	3
<5,9>	7	7	0
<6,9>	12	13	1
<7,9>	7	11	4
<7,10>	7	9	2
<8,10>	13	16	3
<9,10>	14	14	0

10.2 超立方和洗牌网络上的AOE网算法

在第七章我们介绍了 Dekel 等人基于超立方和洗牌网络上的矩阵乘法算法，这里我们介绍他们的计算 AOE 网络拓扑排序和关键路径算法[3]．

10.2.1 超立方洗牌网络上的拓扑排序算法

在超立方和洗牌网络上实现拓扑排序的算法，要充分利用这两个网络上的快速矩阵乘法算法．Dekel 等人的算法正是利用了这一点．

1. 算法的基本原理

假定 $G(V,E)$ 是一个 AOE 网，$V=\{1, 2, \cdots, n\}$．拓扑排序算法的基本思想是：首先计算每对顶点间的最长路径长度．设 G 的每对顶点间的最长路径矩阵用 P 表示．P 的初始值为：

$$P(i,j)=\begin{cases}1, & \text{当} <i,j> \in E \\ 0, & \text{当} i=j \\ -\infty, & \text{当} <i,j> \notin E\end{cases}$$

矩阵 P 可通过计算所有顶点对的最短路径算法获得，但在这里必须用"max"操作替换最短路径算法中的"min"操作．其结果是：对应 P 中元素全为 0 或全为负的行 i_1，i_2，…，i_k，$1 \leqslant i_j \leqslant n$，$1 \leqslant j \leqslant n$，表示顶点 i_1，i_2，…，i_k 没有前趋的顶点．

然后，对 P 的每个不全为 0 或不全为负的行 j_1，j_2，…，$j_l(1 \leqslant j_i \leqslant n$，$1 \leqslant i \leqslant l)$ 分别计算出从顶点 i_1，…，i_k 到达 j_1，j_2，…，j_l 的最长路径长度 l_{j_1}，l_{j_2}，…，l_{j_l}。最后，每个顶点 $i \in V$，形成一个序偶 (l_i, i)，若 $i = i_1$，i_2，…，i_k，则 $l_i \leftarrow 0$；对 $\{(l_i, i)\}_{i=1}^{n}$ 进行非降排序；排序的结果，若某个顶点 i_0 排在第 k 个位置，则顶点 i_0 的拓扑序号就是 k，$1 \leqslant k \leqslant n$。这样就能给每个顶点赋予正确的拓扑序号。因为若 i 是 j 的前趋顶点，则 $l_i < l_j$，所以 (l_i, i) 排在 (l_j, j) 之前，即顶点 i 的拓扑序号小于顶点 j 的拓扑序号。这同拓扑排序的定义是一致的。

2. **算法的非形式化描述**

算法10.5 FINDING TOPOLOGICAL ORDERS ON NETWORKS

输入：G 的邻接矩阵 A；

输出：每个顶点 $i \in V$ 的拓扑序号 $T(i)$，存放在编号为 $(0, 0, i-1)$ 的处理器中。

```
begin
(1)  for each i,j : 1 ≤ i,j ≤ n pardo          / * 赋初值 * /
         if A(i,j) = 1 then P(i,j) ← 1
                   else if i = j  then P(i,j) ← 0   else P(i,j) ← −∞   endif
         endif
     endfor;
(2)  并行计算G的所有顶点间最长路径长度矩阵P;
(3)  for each i,j : 1 ≤ i,j ≤ n pardo
        if P(i,j) ≤ 0 then B(i,j) ← 0   else B(i,j) ← P(i,j)   endif
     endfor ;
(4)  for each i,j : 1 ≤ i,j ≤ n pardo
        P(1,i) ← max {B(i,j) | 1 ≤ j ≤ n}
     endfor;
(5)  for each i: 1 ≤ i ≤ n pardo
          if P(1,i) = 0
                then C(i) ← 1       / * i 没有前趋顶点 * /
                else C(i) ← 0       / * i 有前趋顶点 * /
          endif
     endfor ;
(6)  for each i,j : 1 ≤ i,j ≤ n pardo
        if P(1,i) ≠ 0
          then if C(j) ≠ 0 then Temp(i,j) ← P(i,j) else Temp(i,j) ← −1 endif
          else Temp(i,j) ← −1
        endif
     endfor ;
```

(7) **for** each i,j: $1 \leqslant i,j \leqslant n$ **pardo**

if $P(1,j) \neq 0$ **then** $P(1,j) \leftarrow \max\{\text{Temp}(i,j)\}$ **endif**

endfor;

(8) 对序偶$(P(1,j),j)$, $j=1, 2, \cdots, n$进行非降排序。排序后，若$(P(1,j),j)$在处理器$(0,0,k)$中，则形成序偶$(j,T(j))$，其中$T(j)=k+1$，$0 \leqslant k < n$；

(9) 对形成的序偶排序后，编号为$(0,0,i)$处理器存放着顶点 $i+1$ 的拓扑序号 $T(i+1)$

end.

本节假定超立方或洗牌网络处理器的编号同第七章的编号一致，且假定有 n^3 个处理器.

算法 10.5 的第 (1)～(2) 步是计算所有顶点对之间的最长路径长度矩阵 P；第 (3)～(5) 步是找出所有没有前趋的顶点 i_1，i_2，…，i_k；第 (6)～(7) 步是计算每个有前趋的顶点 j_1，j_2，…，j_l 和从 i_1，i_2，…，i_k 到 j_1，…，j_l 的最长路径长度；第 (8)～(9) 步给顶点赋予拓扑序号并指出这些拓扑序号的存放位置.

定理 10.5 在超立方和洗牌网络上，计算一个 AOE 网 $G(V,E)$，$|V|=n$ 的拓扑排序，算法 10.5 需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器.

证明 算法 10.5 的复杂性分析如下：第 (1) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；根据定理 7.8，第 (2) 步需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器；第 (3) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (4) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (5) 步需 $O(1)$ 时间、$O(n)$ 处理器；第 (6) 步需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (7) 步需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (8)～(9) 步利用 Nassimi 等人的排序算法[11]，在 n^3 处理器的超立方和洗牌网络上，需 $O(\log n)$ 时间. 因此整个算法需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器.

10.2.2 超立方和洗牌网络上的关键路径算法

在 10.1.3 节我们已叙述过，一旦找出 AOE 网的关键活动后，那么，若从源点 s 至终点 t 的路径 L 上所有边的活动都是关键活动，则称 L 为这个 AOE 网的关键路径.

对每个活动 $<i,j> \in E$ 的最早开始时间是从 s 到 i 的所有路径上，其时间权和最大的值作为 i 的最早开始时间. 因而可利用上一节计算最长路径来计算最早开始时间矩阵 EST. EST 的初值定义如下:

$$\text{EST}(i,j)=\begin{cases} W(i,j), & <i,j> \in E \\ 0, & i=j \\ -\infty, & <i,j> \notin E \end{cases}$$

类似地，我们可以得到活动的最迟开始时间矩阵 LST，具体做法是：令 $G'(V,E')$ 是这样一个图，其中 $E'=\{<j,i> \mid <i,j> \in E\}$. 从 t 出发，计算 G' 的所有顶点对的最短路径矩阵 LST1，LST1 初值定义为:

$$\text{LST1}(i,j)=\begin{cases}W(j,i), & <i,j>\in E' \\ 0, & i=j \\ +\infty & <i,j>\notin E'\end{cases}$$

则 $\text{LST}(i,j)=\text{LST}(s,t)-\text{LST1}(i,j)-W(i,j)$，$<i,j>\in E$。所以，每个活动 $<i,j>\in E$ 的松弛时间为：

$$\text{SLACK}(i,j)=\text{LST}(i,j)-\text{EST}(i,j)$$

因此，一个活动 $<i,j>\in E$ 是关键活动当且仅当 $\text{SLACK}(i,j)=0$。

在超立方和洗牌网络上求关键路径算法的非形式化描述如下：

算法 10.6 FINDING CRITICAL PATH ON NETWORKS

输入：G的加权邻接矩阵W；

输出：每个活动 $<i,j>\in E$ 的最早开始时间 $\text{EST}(i,j)$，最迟开始时间 $\text{LST}(i,j)$，松弛时间 $\text{SLACK}(i,j)$，并标识哪些活动是关键活动。

begin

(1) 计算最早开始时间矩阵EST；

(2) 构造辅图$G'(V,E')$；

(3) 计算G'的所有顶点对之间最短路径矩阵LST1；

(4) 计算最迟开始时间矩阵LST；

(5) 计算松弛时间矩阵SLACK；

(6) 判别哪些活动是关键活动

end.

定理 10.6 在超立方和洗牌网络上，计算一个 AOE 网 $G(V,E)$，$|V|=n$ 的关键路径，算法 10.6 需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器。

证明 在由 n^3 个处理器组成的超立方和洗牌网络上，我们来分析算法 10.6 的复杂性。算法的第 (1) 步，由算法 10.5 的第 (1)～(2) 步可知，需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器；第 (2) 步仅需 $O(1)$ 时间、$O(n^2)$ 处理器；第 (3) 步由定理 7.8 可知，需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器；第 (4) 步涉及着先广播 $\text{EST}(s,t)$ 至网络的每个处理器，故这一步至多需 $O(\log n)$ 时间、$O(n^2)$ 处理器；第 (5)～(6) 步显然需 $O(1)$ 时间、$O(n^2)$ 处理器。因此，整个算法需 $O(\log^2 n)$ 时间、$O(n^3)$ 处理器。

10.3 AOE网的分布式算法

Chaudhuri[4] 曾基于消息异步通信的分布式计算模型，对 AOE 网 $G(V,E)$ 建议了一个计算每个事件 $i\in V$ 的最早开始时间、最迟开始时间以及松弛时间的分布式算法。本节

我们介绍这个算法。

假定对每个结点 $i\in V$，定义 $N_{in}(i)=\{u \mid <u,i>\in E\}$ 为 进 入 边 的另 一端点集合，$N_{out}(i)=\{v \mid <i,v>\in E\}$ 为射出边的端点集合。分布式算法分为两个阶段：第一阶段，计算每个结点 $i\in V$ 的最早开始时间 EST(i)；第二阶段，计算每个结点 $i\in V$ 的最迟开始时间 LST(i) 以及松弛时间 SLACK(i)。

10.3.1 算法的形式化描述

在第一阶段开始前，每个结点 i 拥有 $N_{in}(i)$、$N_{out}(i)$ 和 $W(k,i)$，$\forall k\in N_{in}(i)$。在第一阶段终止后，每个结点 i 已经给 EST(i) 赋值。在结点 s 初始化算法中检测第一阶段是否终止执行。

算法10.7 CRITICAL PATH ALGORITHM AT FIRST STAGE

begin

源结点 s 的算法：

初始化：

$n^-(s)\leftarrow|N_{out}(s)|$； EST($s$) $\leftarrow 0$；

send (SET; EST(s)) message to i, for all $i\in N_{out}(s)$；

upon receiving TERM message from node i **do**：

  $n^-(s)\leftarrow n^-(s)-1$；

  if $n^-(s)=0$ **then** terminate Phase Ⅰ **endif**；

一般结点 i 的算法：

初始化：

$n^+(i)\leftarrow|N_{in}(i)|$； $n^-(i)\leftarrow|N_{out}(i)|$； EST($i$) $\leftarrow -\infty$；

upon receiving (SET; d) from node j **do**：

 if EST(i) $< d+W(j,i)$

  then EST(i) $\leftarrow d+W(j,i)$；

    $n^+(i)\leftarrow n^+(i)-1$；

    if $n^+(i)=0$

     then if $N_{out}(i)\neq\phi$

       then send (SET;EST(i)) to k, for all $k\in N_{out}(i)$

       else send TERM message to j, for all $j\in N_{in}(i)$；

        terminate

      endif

    endif

 endif；

```
upon receiving TERM message from node k do:
        n⁻(i) ← n⁻(i) − 1;
        if n⁻(i) = 0 then send TERM message to j, for all j∈N_in(i);
                          terminate
        endif
end.
```

在第二阶段开始前，每个结点 $i\in V$ 有局部变量 $N_{in}(i)$，$N_{out}(i)$，EST(i) 及 $W(i,j)$，$\forall\ j\in N_{in}(i)$。在第二阶段终止时，每个结点 $i\in V$ 拥有 LST(i) 及 SLACK(i) 值，结点 t 起动算法并终止算法的执行。

算法 10.8 CRITICAL PATH ALGORITHM AT SECOND STAGE

```
begin
终点 t 的算法:
初始化:
n⁺(t) ← |N_in(t)|;   LST(t) ← EST(t);   SLACK(t) ← 0;
send (SET; LST(t)) message to node i, for all i∈N_in(t);
upon receiving TERM message from node i do:
      n⁺(t) ← n⁺(t) − 1;
      if n⁺(t) = 0 then terminate Phase II endif;
一般结点 i (≠ t)的算法:
初始化:
n⁺(i) ← |N_in(i)|;   n⁻(i) ← |N_out(i)|;   LST(i) ← ∞;
upon receiving (SET; d) from node j   do:
      if LST(i) > d − W(i,j)
        then LST(i) ← d − W(i,j);
             n⁻(i) ← n⁻(i) − 1;
             if n⁻(i) = 0
               then SLACK(i) ← LST(i) − EST(i);
                 if N_in(i) ≠ φ
                   then send (SET; LST(i)) message to j, for all j∈N_in(i)
                   else send TERM message to k, for all k ∈N_out(i);
                        terminate
                 endif
             endif
```

```
        endif;
    upon receiving TERM message from j do:
        n+(i) ← n+(i) - 1;
        if n+(i) = 0
              then send TERM message to j, for all j∈N_out(i);
                   terminate
        endif
    end.
```

10.3.2 算法的正确性证明

算法在第一阶段的正确性证明基于下列诸引理。

引理 10.4 在有限时间内，每个结点 $i \in V - \{s\}$ 将从每个 $j \in N_{in}(i)$ 收到一个 SET 消息。

证明 我们假定一个结点 $i \in V - \{s\}$ 在从每个 $p \in N_{in}(i)$ 收到 SET 消息后将成为源结点，置 EST(i) 及发送 SET 消息至每一个 $j \in N_{out}(i)$，若 $N_{out}(i) \neq \phi$，假定它已从 G 中删除。源点 s 在起动第一阶段的算法后，置 EST(s) 并发送 SET 消息给每个 $k \in N_{out}(s)$，假定 s 已从 G 中删除。得到的图仍是一个无有向回路的有向图。因此，在该图中至少有一个结点 k 从每个 $q \in N_{in}(k)$ 收到 SET 消息，它成为源结点，然后假定它也从图中删除。根据归纳假定，在有限时间内，每个结点 $i \in V - \{s\}$ 都将在收到每个 $j \in N_{in}(i)$ 发来的 SET 消息，然后成为源结点。如果存在一个结点 $i \in V - \{s\}$，在有限时间内并没有 收 到 所有 $j \in N_{in}(i)$ 发来的 SET 消息，那么至少有一个 $p \in N_{in}(i)$ 没有收到所有 $r \in N_{in}(p)$ 发送的 SET 消息。其结果是，或者 s 没有发送 SET 消息，或者 G 不连通。这两种情况都不存在。故引理成立。

引理10.5 第一阶段是会终止的。

证明 由引理 10.4 可知，在有限时间内，终点 t 收到每个 $j \in N_{in}(t)$ 的 SET 消息后，在置 EST(t) 及发送 TERM 消息至每个 $j \in N_{in}(t)$ 后停止执行。同引理 10.4 类似的讨论，可以证明最终源结点 s 将从每个 $i \in N_{out}(s)$ 收到 TERM 消息，终止第一阶段，故引理成立。

定理 10.7 第一阶段的算法在有限时间内能正确地计算每个结点 $i \in V$ 的最早开始时间 EST(i)。

证明 根据算法 10.7 可知，对每个结点 $i \in V$，若 $i \neq s$，则置 $\text{EST}(i) = \max\{\text{EST}(j) + W(j,i) \mid j \in N_{in}(i)\}$，且 EST($s$) = 0。这同 EST($i$) 的定义是一致的。又由引理 10.4 及引理 10.5 知，第一阶段在有限时间内会终止的。

第二阶段正确性的证明同第一阶段的类似，故不赘述。

定理 10.8 在异步通信的分布式计算模型上，计算一个 AOE 网 $G(V,E)$，$|V| = n$，$|E| = m$ 的关键路径，算法 10.7 及 10.8 需 $O(d)$ 时间，而通信复杂性为 $O(dm)$，其中 d 是图的直径。

证明 对某个结点 i，令 $N_{in}(i) \neq \phi$，$N_{out}(i) \neq \phi$，若每个 $j \in N_{in}(i)$ 发送给 i 的 SET 消息

都引起 EST(i) 值发生改变。这样结点 i 发送的 SET 消息数将是 $|N_{out}(i)|\cdot|N_{in}(i)|$，每个结点 j 的 TERM 消息为 $O(|N_{out}(i)|)$。又从 s 到 t 的最长路径为 $O(d)$，故第一阶段的通信复杂性为 $O(dm)$。又 G 是非有向回路的有向图。因而时间复杂性为 $O(d)$，第二阶段同第一阶段的通信复杂性及时间复杂性在大"O"意义下是一样的。故整个算法的时间复杂性为 $O(d)$，通信复杂性为 $O(dm)$，$1\leqslant d<n$。

10.4 最大流问题的并行算法

Shiloach 等人基于 SIMD—CRCW PRAM 计算模型，曾建议了第一个有向流网络的并行算法[5]。假定计算模型的"同时写"仅是写入"相同的值"时写成功。本节我们介绍这个求最大流问题的并行算法。

10.4.1 有向流网络的基本概念

一个有向流网络 $N=(G,s,t,c)$ 是一个四元组，其中：$G=G(V,E)$ 是一个有向图；s，t 是 G 的两个特殊顶点，它们分别是源点及终点，$s\in V$，$t\in V$；c 是一个 $E\to R^+$ 的函数，R^+ 是正实数域，即每条边 $e\in E$ 都赋有一个非负的容量 $c(e)$。我们用 $d_{in}(v)$ 及 $d_{out}(v)$ 分别表示 G 的顶点 v 的进入边数和射出边数，$v\in V$。

一个函数 f：$E\to R^+$ 是一个流 (Flow)，若它满足：

(a) 容量规则：

$$f(e)\leqslant c(e),\quad \forall e\in E;$$

(b) 守恒规则：

$$\mathrm{IN}(f,v)=\mathrm{OUT}(f,v)\qquad \forall v\in V-\{s,t\};$$

其中：$\mathrm{IN}(f,v)=\sum_{<u,v>\in E} f(u,v)$ 是所有流入 v 的流量之和；

$\mathrm{OUT}(f,v)=\sum_{<v,u>\in E} f(v,u)$ 是所有从 v 流出的流量之和。

一个流 f 的值为：

$$|f|=\mathrm{OUT}(f,s)-\mathrm{IN}(f,s)$$

一个流 f 是最大流(Maximun Flow)，若存在任何其它的流 f'，均有 $|f|>|f'|$。一个流 f 使得一条边 $e\in E$ 满足 $f(e)=c(e)$，则称 f 使边 e 饱和，e 称为饱和边(Staturated Edge)。网络 N 中的一个流是最大的，若从 s 到 t 的每一条有向路径上至少有一条饱和边。

一个有向网络 $N=(G,s,t,c)$ 是一个层次网络，若 G 具有下列性质：

(1) 每个顶点 $v\in V$ 有一个层号 $l(v)$；

(2) $l(s)=0$ 且 $0\leqslant l(v)\leqslant l(t)$，$\forall v\in V$；

(3) 若 $<u,v>\in E$，则 $l(v)-l(u)=1$。

顶点集 $L_j=\{v\mid l(v)=j\}$ 称为 G 的第 j 层，$0\leqslant j\leqslant l(t)$。

10.4.2 最大流并行算法的高层描述

首先我们假定网络 $N=(G,s,t,c)$ 是一个层次网络。这里介绍的最大流算法正是基于这种层次网络。

1. 算法的基本原理

并行算法是由许多"脉冲"步组成的。算法的基本思想是：在第一个脉冲时，源 s 将饱和它的所有射出边。在以后的每个脉冲时，网络 N 的顶点集 $V-\{s,t\}$ 被划分为**平衡顶点** (Balanced Vertices) 和**不平衡顶点**[①] 两个集合。在一个脉冲持续时间内，平衡顶点不做任何工作，但不平衡顶点则试图将尽可能多的额外流量 (Excess Flow) 向前推进。若它们不能按这种方式将所有额外流量消耗掉，则将剩余的额外流量向后返回。向后返回的流量按"后进先出"规则进行。

一个顶点 $v\in V$ 成为**阻塞** (Blocked) 顶点，若它的所有射出边或是饱和边或是进入一个在上一个脉冲时阻塞的顶点。假定终点 t 绝对不会阻塞。

2. 算法的非形式化描述

为便于描述算法，这里定义一些变量如下：

EXCESS(v)：表示从顶点 $v\in V$ 应该向前推进或向后返回的额外流量，它等于IN(f,v) − OUT(f,v)；

AVAILABLE(v)：它是v的射出边的一个子集，由非饱和边以及那些不进入到一个阻塞顶点的边组成；

A_FLOW_QUANTUM(e,q)：表示在某个脉冲，通过边 e 的流量值为q。

要使从一个顶点 v 返回的流量保持"后进先出"规则，我们用一个栈 STACK(v) 记录进入 v 的流量。栈 STACK(v) 内的每个流量用二元组 ($e=\langle u,v\rangle$, q) 形式存贮。

有了上述定义，下面我们给出每个顶点 v 的额外流量向前推进的 PUSH 操作过程以及向后返回的 RETURN 操作过程的细节。

算法10.9 PUSH OPERATION

```
procedure PUSH (v, EXCESS(v));
  begin
    /*  若顶点 v 存在额外流量且可以流出，则执行while循环体。 */
  (1) while EXCESS(v) > 0 and AVAILABLE(v) ≠ φ do
          e ← the first edge of AVAILABLE(v);
             /* 找出第一条可以流出的边e = <v,w> */
          q ← min{c(e) − f(e), EXCESS(v)};   /* 边e可流出的流量值 */
          add Q = (e,q) to STACK(w);
          f(e) ← f(e) + q;    EXCESS(v) ← EXCESS(v) − q;
          if f(e) = c(e) then AVAILABLE(v) ← AVAILABLE(v) − {e}  endif
            /* 若边 e 饱和，则从集AVAILABLE(v)中删除 e  */
```

[①] 一个顶点 v 是平衡顶点，若 IN(f,v) = OUT(f,v)；一个顶点 v 是不平衡顶点，若 IN(f,v) > OUT(f,v)。

```
    endwhile;
(2) if AVAILABLE(v) = Φ   /*若顶点v有多余流量，但不可流出*/
      then block v;
        for each <u,v> ∈E do
            /*删除 v 的前趋顶点时，把<u,v>作为流出边*/
            AVAILABLE(u) ← AVAILABLE(u) − {<u,v>}
        endfor
    endif
end.
```

算法10.10 RETURN OPERATION

```
procedure   RETURN (v, EXCESS(v));
begin
  /*当 v 是阻塞顶点，则它的额外流量需返回流入到 v 的顶点中*/
  while EXCESS(v) > 0 do
      Q = (e = <u,v>, q) ← the first flow in STACK(v);
      q' ← min{q, EXCESS(v)};
      f(e) ← f(e) − q';  EXCESS(v) ← EXCESS(v) − q';
      EXCESS(u) ← EXCESS(u) + q';
      if  q = q'  then delete Q from STACK(v)
                  else Q ← (e, q − q')
       endif
  endwhile
end.
```

有了这两个操作过程，在一个层次网络上找最大流的并行算法可简单地描述为:

算法10.11 FINDING MAX_FLOW ALGORITHM

```
procedure Max_flow(G, s, t, c, f);
 begin
 (1) parallel do   /*L_i是第 i 层顶点集合*/

       EXCESS(s) ← Σ_{v∈L_1} c(s, v);

       PUSH (s, EXCESS(s));
 (2) i ← 1;   /*脉冲计数*/
 (3) while there exist unbalance vertices in N do
         i ← i + 1;
        for each v: v∈V pardo
```

```
                if v is unbalanced
                    then if v isn't blocked
                            then parallel do  PUSH(v, EXCESS(v))
                            else parallel do  RETURN(v, EXCESS(v))
                         endif
                endif
            endfor
        endwhile
    end .
```

下面的引理展示了算法10.11的一个有趣的性质。

引理 10.6 如果 i 是偶（或奇）数，那么在第 i 个脉冲开始时，所有不平衡顶点都在奇（或偶）数层。

证明 对 i 进行归纳。当 $i=1$ 时，s 饱和所有射出边。即第 2 层顶点有可能成为不平衡顶点，引理成立。假设在第 $i(\leqslant k)$ 个脉冲开始时，引理成立。现考虑在第 $j=k+1$ 个脉冲开始时的情况。假定在第 j 个脉冲开始时（不妨设 j 为偶数）有一个不平衡顶点 v 位于偶数层。我们证明这是不可能的。因 j 是偶数，故 $j-1=k$ 是奇数。由归纳假设，在第 $j-1$ 个脉冲开始时，所有不平衡顶点都在偶数层。根据算法可知，在第 $j-1$ 个脉冲时，所有不平衡顶点都将额外流量尽量向前推进一层（即奇数层）。若有剩余的额外流量，则全部向后返回，后一层也是奇数层。结果这个非平衡顶点的额外流量为 0，即变成了平衡顶点。也就是说，在第 $j-1$ 个脉冲之后第 i 个脉冲开始时，所有不平衡顶点都在奇数层，产生矛盾。故引理成立。

应该指出的是：在一个层次网络上发现了最大流之后，下一步就应构造一个新的层次网络。具体来讲，可分为两阶段完成。第一阶段是构造一个新的网络，这个新网络含有边 $e=<u,v>$ 当且仅当下列一个条件成立：

(1) $f(e)<c(e)$，边 e 的新的容量 $c'(e)=c(e)-f(e)$；

(2) $f(v,u)>0$，边 $<v,u>$ 的新容量是 $f(e)$。

第二阶段，从源点 s 开始，对新的网络执行宽度优先搜索，产生一个层次网络，可采用第三章介绍的并行宽度优先搜索法来实现。

3. 算法的正确性验证

要证明最大流并行算法 10.11 是正确的，只要证明在每个层次网络，算法 10.11 都产生一个最大流就足够了。因此，下面的证明都是基于层次网络。

定理 10.9 (*a*)在层次网络中，至多要 $2l(n+1)$个脉冲，算法 10.11 就会终止；其中 l 是网络的层数。(*b*)当算法 10.11 终止时，该算法产生一个最大流。

证明 (*a*)因为在经过 $2l$ 个连续的脉冲后，若不存在阻塞顶点，则所有顶点 $v\in V-\{s,t\}$的额外流量都已分别流入源点 s 或终点 t 中。即所有顶点都成为平衡顶点，故算法 10.11 终止。

(*b*)当算法 10.11 终止时，所有顶点都成为平衡顶点。令 $P=u_0u_1u_2\cdots u_l$ ($u_0=s$, $u_l=t$) 是

任意一条从 s 到 t 的有向路径，我们证明 P 上至少有一条饱和边。令 $u_k(0 \leq k < l)$是 P 上距离 t 最近的一个阻塞顶点。则边$<u_k, u_{k+1}>$是一条饱和边。在 u_k 被阻塞的那个脉冲，边$<u_k, u_{k+1}>$一定是饱和边。因为 u_{k+1} 没有阻塞，这条边一直处于饱和状态。又因为 u_{k+1} 绝不阻塞，因面也绝对没有返回的流量.

10.4.3 PS－树及其操作

在描述最大流并行算法的实现细节之前，我们介绍一种 PS－树以及在这种树上的操作. PS－树是本节并行算法使用的主要数据结构. 现已证明 PS－树在栈、队列以及向量维护方面非常有用. 在并行算法设计中，它是一种很有前途的工具.

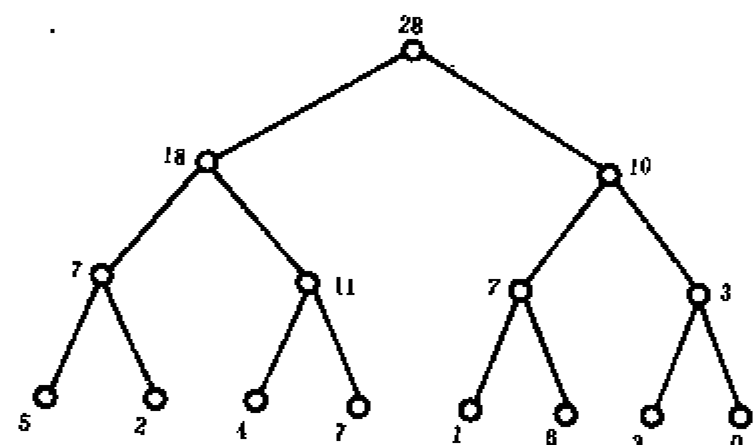


图 10.3 PS—树示例($n=7$)

已知 k 个元素 a_1, a_2, …, a_k 我们把它们同一棵完全二叉树联系起来，这棵树记为 $T(a_1, a_2, \cdots, a_k)$叫做 PS－树(Partial Sums Trees)。PS—树有 $2^{\lceil \log k \rceil}$个叶子。最左的 k 个叶子称为活动叶子，它们分别对应 a_1, a_2, …, a_k。其它叶子赋值 0. 每个非叶结点 x 又是一棵完全二叉树 T_x, x 是 T_x 的根，它的值是(i_2-i_1+1)个元素 a_{i_1}, a_{i_1+1}, …, a_{i_2} 之和，其中: a_{i_1}, a_{i_1+1}, …, a_{i_2} 是 T_x 的叶子. 图 10.3 显示了一棵 PS—树 $T(5, 2, 4, 7, 1, 6, 3)$。

下面我们介绍 PS－树上的一些基本操作. 设 T 是一棵 PS－树. T 的每个结点用 $T[h,i]$ 表示，这里 h 是结点 $T[h,i]$ 在树中的高度（约定叶子的高度为 1)，i 是结点在同一高度上其它结点间的顺序号，$h(T)$代表 T 的高度. 具体的例子如图 10.4 所示。

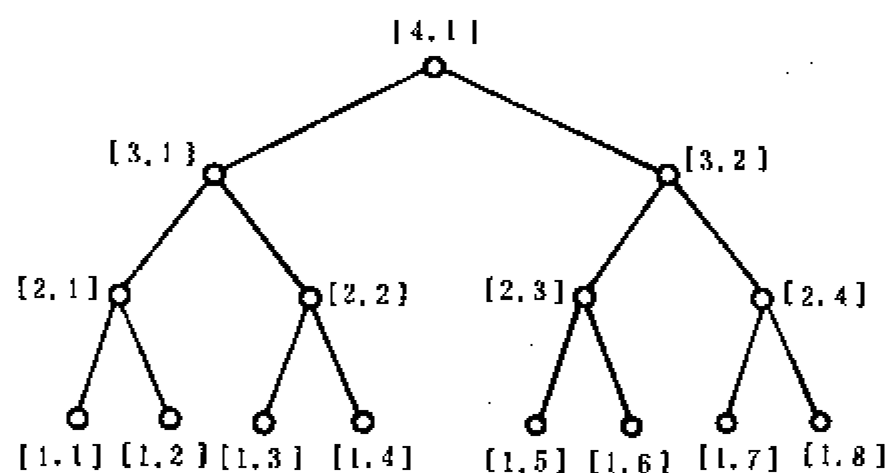


图 10.4 用高度表示的 PS—树示例

为了便于描述，假定树 T 的每个叶子 $T[1,i]$ 赋有一个编号为 i 的处理器。

(1) CLEAR 操作

CLEAR(i,T) 操作的功能是：编号为 i 的处理器将从 $T[1,i]$ 到树根 $T[h(T),1]$ 路径上的所有结点赋值为 0. 显然，若有多个处理器在同一棵树上执行 CLEAR 操作时，则可能引起同时写"0"冲突.

算法10.12 CLEAR ON PS—TREE

```
procedure CLEAR(i,T);
  begin
        for j←1 to h(T) do
              T[j, ⌈i/2^(j-1)⌉]←0
```

        endfor

    end .

(2) UPDATE 操作

在 UPDATE(i, a_i, T) 操作中，树的第 i 个叶子的值赋为 a_i。这将引起其它一些结点值的改变，同时可能存在多个这样的改变。

算法10.13 UPDATING ON PS—TREE

```
procedure UPDATE(i, a_i, T);
  begin
  (1)  T[1,j] ← a_i ;
  (2)  for j ← 2 to h(T) do
          T[j, ⌈i/2^(j-1)⌉] ← T[j-1, 2⌈i/2^(j-1)⌉-1] + T[j-1, 2⌈i/2^(j-1)⌉]
       endfor
  end .
```

(3) SUM 操作

SUM(i, s_i, T) 的功能是：$s_i \leftarrow a_1 + a_2 + \cdots + a_i$。

算法10.14 SUMMING ON PS—TREE

```
procedure SUM(i, s_i, T);
  begin
  (1)  s_i ← a_i ;
  (2)  for j ← 2 to h(T) do
          if 2⌈i/2^(j-1)⌉ = ⌈i/2^(j-2)⌉  then s_i ← s_i + T[j-1, ⌈i/2^(j-2)⌉-1]  endif
       endfor
  end .
```

(4) FIND 操作

FIND(α, k, ρ, T) 的功能是：对给定的 α，FIND 返回 k 及 ρ，使得：

$$a_1 + a_2 + \cdots + a_{k-1} < \alpha_1 \leqslant a_1 + a_2 + \cdots + a_k \quad 且 \quad \rho = \alpha - \sum_{i=}^{k-1} a_i。$$

算法10.15 FINDING ON PS—TREE

```
procedure FIND(α, k, ρ, T);
  begin
  (1)  k ← 1 ;   ρ ← α ;
  (2)  for j ← h(T) downto 2 do
```

```
            if ρ > T[j-1, 2k-1]
             then   ρ ← ρ - T[j-1, 2k-1];      k ← 2k
             else   k ← 2k - 1
            endif
        endfor
    end .
```

10.4.4 最大流算法的并行实现

上一小节我们介绍了 PS— 树以及它上面的一些基本操作。这里将详细介绍最大流算法的并行实现。

首先我们介绍在实现最大流算法时需用到的一些数据结构。

网络的每个顶点 $v \in V$ 有四种不同的树。它们分别是:

(1) T_OUT(v): 这棵树有$d_{out}(v)$个活动叶子，每个叶子代表v的一条射出边。叶子的值是射出边向前推进（即从v流出）的流量;

(2) T_IN(v): 这棵树有$2n \times d_{in}(v)$个活动叶子($2n$代表脉冲数目)，它模拟STACK(v)的功能。记录在T_IN(v)树中（从左至右）叶子上的流量，同它们在栈中记录的次序一致;

(3) T_ACCESS(v): 这棵树有$d_{in}(v)$个活动叶子。每个叶子代表v的一条进入边，它协调多个处理器同时修改STACK(v)的活动;

(4) T_SUM(v): 这棵树有$d_{out}(v)$个活动叶子，每个叶子对应v的一条射出边。在某一脉冲时，用它计算返回到v的流量总和。

网络的每条边 $e \in E$ 也对应有一棵 PS— 树，记作 T_EDGE(e)。这棵树有 $2n$（脉冲数目）活动叶子，每个叶子对应一个脉冲。在某一脉冲时，用它计算返回到 e 上的流量和。

要实现最大流并行算法，关键是 PUSH 操作以及 RETURN 操作的并行实现。应该注意的是：在算法的并行实现过程中，在每个脉冲内，PUSH 操作执行完之前不能执行 RETURN 操作。

为简化描述，假定每个顶点 $v \in V$ 分派一个处理器 PE(v)。每条边 $e \in E$ 也分派一个处理器 PE(e)。此外，T_IN(v) 每个叶子也分派一个处理器，且流量为 Q 的叶子处理器记作 PE(Q)。

在整个算法开始之前，置所有的PS－树叶子值为0。

若把形成一个新的层次网络，并找出新的网络上最大流过程称为一个阶段，则算法 10.16 用于每阶段的初始化工作。

算法 10.16　INITIALIZATION OF MAX－FLOW PROBLEM

```
  procedure INITIALIZE;
    begin
```

PE(e_j = < v,w >) **do** :

(1.1) **call** UPDATE (j, $c'(e)$, T_OUT(v));

(1.2) $f(e_j) \leftarrow 0$;

PE(v) **do** :

(1.3) hd(v) ← 0 ; $k'(v) \leftarrow 1$

endfor

end .

在算法 10.16 中，第 (1.1) 步的 j 是 v 的第 j 条射出边，即对应 T_OUT(v) 的第 j 个叶子，c' 是当前层次网络边 e 上的容量。第 (1.3) 步 hd(v) 是栈 STACK(v) 的栈顶指针，即指向 T_IN(v) 最右一个有用叶子．$k'(v)$ 是存放在 AVAILABLE(v) 内边的最小下标，即 $k'(v) = \min\{j \mid e_j \in \text{AVAILABLE}(v)\}$.

今后我们用 T[root] 表示树 T 的根值，即 T[root] 是 $T[h(T),1]$ 的值.

下面我们给出 PUSH 和 RETURN 操作过程的详细实现算法，并简称为 PUSH 操作和 RETURN 操作，现给出 PUSH 操作如下:

算法10.17 PUSH OPERATION IMPLEMENT

procedure PUSH(v, EXCESS(v));

begin

(1) PE(v) **do** :

(1.1) $\alpha \leftarrow \min\{\text{EXCESS}(v), \text{T_OUT}(v)[\text{root}]\}$;

(1.2) EXCESS(v) ← EXCESS(v) − α ;

(1.3) **call** FIND (α, $k(v)$, ρ, T_OUT(v));

(2) PE(e_j = < v,w >) **do** :

(2.1) **if** $k'(v) \leqslant j \leqslant k(v)$ /* $e_j \in$ AVAILABLE(v) */

then /* r 是树T_ACCESS(w)中对应 e_j 叶子的下标 */

(2.1.1) **call** UPDATE(r, 1, T − ACCESS(w));

(2.1.2) **call** UPDATE(r, S_r, T − ACCESS(w));

(2.1.3) **if** $j \neq k(v)$ **then** $q_j \leftarrow$ T − OUT(v)[1,j] **else** $q_j \leftarrow \rho$ **endif** ;

(2.1.4) $f(e_j) \leftarrow f(e_j) + q_j$;

(2.1.5) TOTAL(w) ← T_IN(w)[root];

(2.1.6) **call** UPDATE(hd(w) + S_r, q_j, T_IN(w));

(2.1.7) **call** UPDATE(j,T_OUT(v)[1, j] − q_j, T_OUT(v));

(2.1.8) hd(w) ← hd(w) + T_ACCESS(w)[root];

(2.1.9) **call** CLEAR(r, T_ACCESS(w));

(2.1.10) EXCESS(w) ← T_IN(w)[root] − TOTAL(w)

```
        endif;
    (3) k'(v) ← k(v);
    (4) PE(e_l = <u,v>) do:
            if EXCESS(v) > 0
                then BLOCK(v) ← "yes";
                    call UPDATE(j, 0, T_OUT(u))
            endif
  end.
```

在算法 10.17 中，第 (1) 步 T_OUT(v)[root] 表示从 v 应该流出的流量．α 是当前脉冲可以流出的量。第 (1.3) 步是处理器 PE(v) 找从边 $e_{k'(v)}$，$e_{k'(v)+1}$，…，$e_{k(v)}$ 流出的流量 α，其结果是边 $e_{k'(v)}$，$e_{k'(v)+1}$，…，$e_{k(v)-1}$ 将成为饱和边，α 的剩余量 ρ 从边 $e_{k(v)}$ 流出．第 (2.1.1)～(2.1.2) 步是有多个处理器试图加入流量到 STACK(w) 中，S_r 是处理器 PE(e_j) 存取 STACK(w) 的顺序号；第 (2.1.3)～(2.1.4) 是将 v 的额外流量一部分通过边 $e_{k'(v)}$，…，$e_{k(v)}$ 流出，这些边的流量分别为 $q_{k'(v)}$，$q_{k'(v)+1}$，…，$q_{k(v)}=\rho$；第 (2.1.5) 步 TOTAL(w) 是到目前为止，已流入 w 的流量总和；第 (2.1.6) 步的 hd(w) + S_r 是对应流量 (e_j,q_j) 的 T_IN(w) 树叶下标，这个流量被记录在栈 STACK(w) 中，且树 T_IN(w) 作适当的修改以便记录流量 (e_j, q_j)；第 (2.1.7) 步对 e_j 的剩余容量进行修改；第 (2.1.8) 步更新栈顶指针，在当前脉冲内，T_ACCESS(w)[root] 含有进入 STACK(w) 流量的个数；第 (2.1.9) 步则清除 T－ACCESS(w) 以供下一个脉冲时使用；第 (2.1.10) 步重新计算 w 处的额外流量．第 (3) 步表示 AVAILABLE(v) 是目前剩下的边中的最小下标；第 (4) 步 l 是树 T_OUT(u) 的第 l 个叶子的下标，这个叶子的对应边为 e_l。由于 e_l 进入一个阻塞顶点 v，故根据 AVAILABLE(u) 定义，e_l 必须删除掉。

现在我们介绍 RETURN 操作过程的实现．

算法 10.18　RETURN OPERATION IMPLEMENT

```
 procedure RETURN(v, EXCESS(v));
  begin
    (1) PE(v) do:
        (1.1) call FIND(T_IN(v)[root] − EXCESS(v), k(v), ρ, T_IN(v));
        (1.2) EXCESS(v) ← 0;
    (2) PE(Q_j = (e_j = <u, v>, q_j)) do:
        (2.1) if k(v) < j ≤ hd(v)
                  then d_j ← q_j
                  else if j = k(v) then d_j ← q_j − ρ endif
              endif;
```

```
(2.2)  if k(v) < j ≤ hd(v)
           then call UPDATE(j ,0, T_IN(v))
           else if j = k(v) then UPDATE(j, ρ, T_IN(v))   endif
       endif ;
       if k(v) ≤ j ≤ hd(v) then
(2.3)      call UPDATE (r_j, d_j, T_EDGE(e_j)) ;
(2.4)      f(e_j) ← f(e_j) − T_EDGE(e_j)[root] ;
(2.5)      call UPDATE(l_j, T_EDGE(e_j)[root], T − SUM(u)) ;
(2.6)      EXCESS(u) ← EXCESS(u) + T_SUM(u)[root] ;
(2.7)      call CLEAR(r_j, T_EDGE(e_j)) ;
(2.8)      call CLEAR(l_j, T_SUM(u))
     endif ;
(3)  PE(v) do :  hd(v) ← k(v)
end .
```

在算法 10.18 中，从 v 返回的流量涉及取消 STACK(v) 里适当的几个流量．算法的第 (1.1) 步就是确定还有多少流量在 STACK(v) 中，其余的流量将在第 (1.2) 步被删除掉；第 (2.1)～(2.2) 步涉及对栈 STACK(v) 进行适当的修改，即从 STACK(v) 中去掉值为 EXCESS(v) 的流量；第 (2.3) 步的 r_j 是 Q_j 从 STACK(v) 被取出的那个时刻的脉冲；它也是 T_EDGE(e_j) 树叶下标，这个叶的对应脉冲为 r_j；T_EDGE(e_j)[root] 计算在第 r_j 个脉冲时，边 e_j 上返回的流量总和；第 (2.4) 步修改边 e_j 上的流量值；第 (2.5) 步的 l_j 是 T_SUM(u) 中树叶的下标，它和边 e_j 相对应；第 (2.6) 步更新 u 的额外流量值，T_SUM(u)[root] 存贮着当前脉冲返回到 u 的流量总和；第 (2.7)～(2.8) 步清除 T_EDGE(e_j) 及 T_SUM(u)，以便下一个脉冲到来时使用．第 (3) 步修改 STACK(v) 的栈顶指针。

在每一阶段，我们还要清除 T_OUT(v) 及 T_IN(v) 内容，以便下一阶段使用，可使用下述过程来完成。

算法10.19　CLEAN OPERATION

```
procedure CLEAN ;
  begin
  (1)  for each v: v∈V   pardo
    (2.1)  PE(e_j = <v,w>) do:  call CLEAR(j, T_OUT(v)) ;
    (2.2)  PE(Q_j = <u,v>, q_j)  do :
                if 1 ≤ i ≤ hd(v) then  call CLEAR(i, T_IN(v))  endif
       endfor
```

end .

引理 10.7 在 SIMD – CRCW PRAM 模型上，算法 10.17 及算法 10.18 均需 $O(\log n)$ 时间、$O(nm)$ 处理器.

证明 算法 10.17 和 10.18 都是在高度至多为 $O(\log nm)=O(\log n)$ 的树上进行操作，显然只需 $O(\log n)$ 时间. 在算法 10.17 中，每个顶点 $v\in V$ 的 T__IN(v) 树需 $2n\times d_{in}(v)$ 个处理器。故所有顶点的 T__IN 树需 $\sum_{v\in V}2n\times d_{in}(v)=2nm$，即算法 10.17 需 $O(nm)$ 处理器. 同样，算法 10.18 也需 $O(nm)$ 处理器.

10.4.5 最大流算法的有效实现

上面描述的最大流算法是一个使用 $O(nm)$ 处理器的实现算法. 本节我们将给出一个有效的并行实现算法. 新的算法通过对处理器小心合理地分配，得出一个时间复杂性相同但只需 $O(n)$ 处理器的最大流并行算法.

首先，我们介绍 Brent 定理[6]，有效的实现算法正是基于这个定理的.

定理 10.10. 对任何一个同步并行算法，若在 t 个时间内完成 x 个基本操作步，则用 p 个处理器可以在 $\lceil x/p\rceil+t$ 个时间步完成.

证明 在同步并行算法中，令 x_i 表示在第 i 个时间步完成的基本操作步 $(1\leqslant i\leqslant t)$，即 $\sum_{i=1}^{t}x_i=x$. 现有 p 个处理器，对每个时间步完成的基本操作步 x_i，用它们模拟时需 $\lceil x_i/p\rceil$ 时间步. 因此，用 p 个处理器完成 x 个基本操作步需时间步为:

$$\sum_{i=1}^{t}\lceil x_i/p\rceil\leqslant\sum_{i=1}^{t}(x_i/p+1)\leqslant\lceil x/p\rceil+t$$

在给出有效的实现算法之前，我们分析算法 10.17 和 10.18 所执行的基本操作步数. 因为 PUSH 和 RETURN 操作都涉及到顶点和脉冲序号 $i(1\leqslant i\leqslant 2n)$ 共有 $O(n^2)$ 个基本操作，这意味着在一个层次网络中找最大流需 $O(n^2\log n)$ 基本操作，而今有 $O(nm)$ 处理器，即每个顶点都同时执行 PUSH 及 RETURN 操作. 故在一个层次网络中找最大流需 $O(n\log n)$ 时间，而求原始网络最大流需构造 n 个这样的层次网络. 故算法的基本操作步数为 $O(n^3\log n)$，时间步数为 $O(n^2\log n)$.

利用定理 10.10，我们将给出一个在 $O(n)$ 处理器上模拟算法 10.17 和 10.18 的并行算法，但时间复杂性不变。为此，引进 T__ASSIGN 树，它是一棵 PS – 树，T__ASSIGN 有 n 个活动叶子，每个叶子对应网络的一个顶点，它将帮助我们解决定理 10.10 的处理器分配问题.

为了执行任意顶点 PUSH 操作或 RETURN 操作的一步，在执行之前我们需要知道所用的处理器数目. 对任意顶点 $v\in V$，PUSH 操作及 RETURN 操作的第 (1) 步和第 (3) 步仅需 $O(1)$ 处理器; PUSH 及 RETURN 操作的第 (2) 步分别需 $(k(v)-k'(v)+1)$ 个处理器和 $(\mathrm{hd}(v)-k(v)+1)$ 个处理器; PUSH 操作第 (4) 步至多需 $d_{in}(v)$ 处理器; 实现每个阶

段初始化的算法 10.16 和算法 10.19 所需要的处理器数目同 PUSH 操作及 RETURN 操作也是一致的.

下面，算法 10.20 利用 T_ASSIGN 树，只用 n 个处理器就可模拟它们. 即把算法 10.20 分别应用在算法 10.16 到算法 10.19 的每一步中。假定处理器及顶点的编号都是 1 到 n，还假定顶点 j 执行当前一步需 $N(j)$ 个处理器。

算法 10.20 **SIMULATING ON SIMD MACHINE**

```
procedure SIMULATE;
  begin
  (1) for each j : 1 ≤ j ≤ n pardo
     (1.1) call CLEAR(j, T_ASSIGN);
     (1.2) call UPDATE(j, N(j), T_ASSIGN);
     (1.3) k(j) ← 1;      / * x = T_ASSIGN[root] * /
     (1.4) while j + (k(j) − 1)n ≤ T_ASSIGN[root] do
                  g1(j) ← j + (k(j) − 1)n ;
                  call FIND(g1(j), j, g2(j), T_ASSIGN);
                  执行j的第g2(j)个处理器当前步的工作;
                  k(j) ← k(j) + 1
            endwhile
        endfor
  end .
```

10.4.6 算法的复杂性分析

为了证明在一个层次网络中，求最大流的并行算法只需 $2n$ 个脉冲就可终止，首先我们证明一些引理. 为叙述方便，我们定义一个所谓合法的三元组 $(e = <u,v>,\ i_p,\ i_r)$. 若存在一个流量 $Q = (e,q)$，在第 i_p 个脉冲时，它通过边 e 流入 v，且记录在 STACK(v) 中。但在第 i_r 个脉冲时，某个返回到 v 的流量引起 Q 的改变（此刻 Q 位于 STACK(v) 的栈顶位置）. 由引理 10.6 可知，若 $(e\ ,i_p\ ,i_r)$ 是一个合法的三元组，那么 i_p 为奇数当且仅当 i_r 为偶数。

引理 10.8　设 $(e = <u,v>,\ i_p,\ i_r)$ 是一个合法的三元组且 $i_p < i < i_r$，那么存在一顶点 w 及整数 i_{p1}，i_{r1}，使得 $(e_1 = <v,w>,\ i_{p1},\ i_{r1})$ 是一个合法的三元组且 $i_{p1} \leqslant i \leqslant i_{r1}$.

证明　假设存在一个 i_0 使得 $i_p < i_0 < i_r$，但引理条件不能满足. 令 i_b 是使 v 阻塞的脉冲序号，$i_p \leqslant i_b \leqslant i_r$. 我们考虑两种情形.

情形1: $i_0 \geqslant i_b$

情形1: $i_0 \geqslant i_b$

因为 i_0 满足 $i_p < i_0 < i_r$，所以 $i_r - i_p > 1$. 这说明在第 i_r 个脉冲时，从 v 返回的某个流量改变了 STACK(v) 的一个流量 Q，而 Q 是在第 $i_r - 1$ 个脉冲之前推入 STACK(v) 的. 这意味着在第 $i_r - 1$ 个脉冲时，有某个流返回到 v，因此的确存在一个三元组 ($<v, w>$, i_x, $i_r - 1$). 在第 i_x 个脉冲时，某个流量从 v 流出且 $i_x \leqslant i_b$，因此 $i_x \leqslant i_0 \leqslant i_r - 1$，产生矛盾.

情形2: $i_0 < i_b$

在这种情形下，在第 i_p 个脉冲结束之前，任何流入 v 的流量不能从 v 返回. 因此，从 v 返回的总流量不超过在第 i_p 个脉冲之后流入 v 的总流量. 这与存在一个三元组 ($<u, v>$, i_p, i_r) 矛盾.

令 q 是某一时钟脉冲 i_1 时流入 v 的流量. $i_1 \leqslant i_p$，在第 $i_1 + 1$ 个脉冲时，这个流量 q 从 v 流出. 令 e_1 是 q 的流出边. 若不存在合法的三元组 (e_1,$i_1 + 1$,i_2)，则在第 $i_1 + 1$ 个脉冲时，通过 e_1 的流量再不会返回. 如果存在一个这样的三元组，那么 $i_2 < i_0 < i_b - 1$. 因此，这个三元组对应的流量再也不会从 v 返回. 这意味着存在几条边，在第 $i_2 + 1$ 个脉冲时，这个流量 q 从 v 流出且是通过这几条边流出的. 同上面的讨论一样，在这个脉冲中通过这几条边的任何流量或者根本不会返回、或者在第 i_0 个脉冲之前返回到 v 并重新寻找路径流出.

引理 10.9　设 v_1，v_2，…，v_k 是第 j_0 层 L_{j_0} 的 k 个阻塞顶点，

$$(e_1 = <u_1, v_1>, i_{p_1}, i_{r_1}), \cdots, (e_k = <u_k, v_k>, i_{p_k}, i_{r_k})$$

是合法的三元组，令

$$A_{j_0} = \{a \mid a \text{ 是偶数且 } i_{p_b} \leqslant a \leqslant i_{r_b}, \quad \text{对某个 } 1 \leqslant b \leqslant k\},$$

那么，在算法终止时第 j 层 L_j 阻塞的顶点数至少是 $|A_{j_0}|$，$j_0 \leqslant j < l(t)$，$l = l(t)$。

证明　对 $k = l - j_0$ 进行归纳。当 $k = 1$ 时，$j_0 = l(t) - 1$. 由于 L_{j_0} 是最接近于 t 的一层，在某个脉冲时推入此层的流量或者在下一个脉冲返回，或者根本就不返回. 因此，$i_{r_b} = i_{p_b} + 1$，$1 \leqslant b \leqslant k$. 即是说 $|A_{j_0}| \leqslant k$. 然而由于 v_1，…，v_k 都有流量返回，所以它们才都是阻塞的.

假定引理对 L_{j_0+1} 成立，现证明它对 L_{j_0} 也成立. 设 v_1，…，v_k 是 L_{j_0} 的 k 个阻塞顶点，且 ($<u_1, v_1>$, i_{p_1}, i_{r_1})，…，($<u_k, v_k>$, i_{p_k}, i_{r_k}) 是合法的三元组. 根据引理 10.8，在 L_{j_0+1} 存在 g (g 为整数) 个阻塞顶点 w_1，w_2，…，w_g 和合法的三元组 ($<v_{d_1}$,$w_1>$, i'_{p_1}, i'_{r_1})，…，($<u_{d_g}$, $w_g>$, i'_{p_g},i'_{r_g}) 且满足

$$\bigcup_{a=1}^{k}[i_{p_a}+1, i_{r_a}+1] \subset \bigcup_{b=1}^{g}[i'_{p_b}, i'_{r_b}]^{①}$$

因此，$|A_{j_0+1}|+k \geq |A_{j_0}|$.

实际上，有可能存在多个三元组进入一个已知顶点 $w \in \{w_1, \cdots, w_g\}$．然而，若 $(<v_\alpha, w>, i_p, i_r)$ 及 $(<v_\beta, w>, i'_p, i'_r)$ 都是合法的三元组，那么栈 STACK(w) 的"后进先出"次序蕴含着或者 $[i_p, i_r] \subseteq [i'_p, i_r']$，或者 $[i'_p, i_r'] \subseteq [i_p, i_r]$．因此对每个 w，选择了最大区间．由归纳假设，到算法终止时，在 L_j 里至少有 $|A_{j_0+1}|$ 个阻塞顶点，$j_0+1 \leq j < l(t)$．除了这些阻塞顶点外，L_j 里至少还有 $|A_{j_0}|$ 个阻塞顶点，$j_0 \leq j \leq l$.

定理 10.11　在层次网络中，算法 10.11 至多需 $2n$ 个脉冲，算法将终止.

证明　假定算法 10.11 在经过 $2n$ 个脉冲之后仍不终止，那么，在 $2n$ 个脉冲后，网络中至少还存在一个不平衡顶点 v。令 E' 是这样一些边 e 的集合。在经过 $2n$ 个脉冲后，e 既不是饱和边又不是进入到一个阻塞顶点的进入边．通过对每条边 $e \in E'$ 赋新的容量 $c'(e)=f(e)$ 来修改层次网络．算法 10.11 在修改后的网络上、同它在原来网络上同样地执行，直至第 $2n$ 个脉冲结束。然而，在修改后的网络中所有额外流量将返回到 s，这是不可能的。假定在第 i 个脉冲 $(i>2n)$ 时，一个流量从 u 返回到 s．因而存在一个合法的三元组 $(<s,u>, l, i)$，根据引理 10.9，阻塞的顶点数 $\geq |A_1| > n$，产生矛盾.

定理 10.12　在 SIMD－CRCW PRAM 上，计算一个网络 $N=(G, s, t, c)$ 的最大流算法，需 $O(n^2\log n)$ 时间、$O(n)$ 处理器.

证明　并行算法 10.11 有 $x=O(n^3\log n)$ 个基本操作步，时间步 $t=O(n^2\log n)$．应用定理 10.11 知，可使用 $O(n)$ 个处理器模拟，模拟时间为：$O(\lceil x/p \rceil + t)=O(n^2\log n)$.

10.5　最大流问题的分布式算法

Cheung 基于著名的 Ford－Fulkerson 算法，用第三章的深度优先搜索技术对最大流问题建议了一个分布式算法[7]．本节将介绍这个算法.

10.5.1　最大流问题的一些基本概念

令 $\alpha(u)$ 及 $\beta(u)$ 分别表示顶点 $u \in V$ 的射入边集合和射出边集合．不失一般性，假定边 $<u,v> \in E$，那么 $<v,u> \in E$，但 $<v,u>$ 的容量可能为 0．令 $\delta<u,v> = c(u,v)-f(u,v)+f(v,u)$ 是边 $<u,v>$ 及边 $<v,u>$ 沿 $<u,v>$ 方向的吞吐容量(Throug put　Capacity)。若 $\delta(u,v)>0$，则 v 是从 u 可达的．若 $\delta(u,v)=0$，则边 $<u,v>$ 是一条饱和边.

①记号 [c,d] 代表集合{c,c+1,c+2,⋯,d}.

根据当前的流 f，我们定义标号过程及扫描 (Scanning) 过程如下：所谓标号 (Labeling) 一个顶点 $v \in V$，是指赋予 v 一个二元组 (u,ε)，其中顶点 u 是将它当前的流量流入 v，ε 表示从源点 s 出发经过 v 通过边 $<u,v>$ 的最大流量。所谓**扫描** (Scanning) 一个顶点 u，是指尚未标号的 u 对一些顶点或对所有顶点进行标号，这些顶点都是 u 的后继顶点，仅当标号过的顶点才有可能成为扫描顶点。

Ford – Fulkerson 算法是一个迭代过程。根据当前的流 f，每次迭代开始时，首先给源点 s 标号 (s^+, ∞)，然后，某个标号顶点就试图扫描和标号其它顶点，直至出现下列某一种情形后才终止目前的标号过程:

(i) 终点 t 被标号。此时，找到一条增广路径 P，对路径 P 上所有边 $<u,v>$ 的流量 $f(u,v)$ 增加一个流量 δ_{min}，其中 $\delta_{min} = \min\{\delta(u,v) \mid <u,v>$ 是 P 上的边 $\}$。然后废弃所有顶点的标号，从源点 s 开始进入下一次迭代。

(ii) 除终点 t 未标号外，所有标号顶点均已扫描过。在这种情况下，当前流 f 即为最大流，算法终止执行。

在串行算法中，每次迭代结束时，必须废弃上一次迭代时的标号。然而，在分布式计算环境里，若不能在每次迭代结束后废弃顶点标号，将造成不同迭代之间标号的二义性。为了消除标号的二义性，在每次迭代后可广播一消除标号的消息，显然这样会增加系统的开销。为避免这种开销，我们将每次迭代序号也包括在所有标号消息中广播。

在给出分布式算法之前，我们约定一些记号如下:

(1) R_r 表示顶点 r 的当前流可达的顶点集，即 $R_r = \{v \mid <r,v> \in E$ 且 $\delta(r,v) > 0\}$。

(2) L_r 表示本次迭代过程 r 还未送标号消息给 L_r 内的元素，且 $L_r \subseteq R_r$。

(3) $P = \{d, u, v, t; \varepsilon\}$ 表示流量增广路径的一部分，它由顶点 d，u，…，v，t 组成，且通过 P 的最大流量为 ε。

(4) (path;P)表示一个路径消息，它通知接收者 r，增广路径已扩充到 r，即 $P \leftarrow P \cup \{r\}$。

10.5.2 深度优先搜索的最大流问题的分布式算法

算法10.21 FINDING MAX – FLOW USING DFS TECHNIQUE

输入：每个结点 $r \in V$ 的邻居集 $N_r = \{v \mid <r,v> \in E\}$ 及容量集 $c(r,u)$，对每个 $u \in N_r$；

输出：源点 s 存放着最大流中各条流的增广路径。

/ * 算法开始时，从源点 s 开始用第三章的分布式深度优先搜索算法遍历图 $G(V,E)$，每个遍历过的顶点被标号为(0,0,0)，同时置 $\delta(r, u) \leftarrow c(r,u)$，$\forall u \in N_r$，$R_r \leftarrow \{v \mid v \in N_r$ and $\delta(r, v) > 0\}$ 。* /

begin

源点 s 处的算法:

$i_s \leftarrow 1$;　　/ * 置迭代序号 * /

```
send (label, i_s, s, ε) message to v, v∈R_s,  ε = max {δ(s, v) | v∈R_s} ;
upon receiving (label, i, d, ε) from node d do:
    if i = i_s  then send (echo, s) to d endif;  / * echo是应答消息 * /
upon receiving (echo, d) from node d do:  The algorithm terminates.
upon receiving (path; P) from d do:
    P ← {s}∪P ;
    δ(s, d) ← δ(s, d) − ε ;
    if δ(s, d) = 0 then R_s ← R_s − {d} endif ;
    store P in node s ;
    if R_s ≠ φ  then i_s ← i_s + 1 ;
                  L_s ← R_s ;
                  select v∈L_s ;
                  L_s ← L_s − {v} ;
                  send (label, i_s, s, δ(s,v)) to v
              else The algorithm terminates
    endif ;
终点 t 处算法:
/ * 一旦从 v 收到 (label, i, d, ε) 消息，结点 t 开始构造流的增广路径 * /
upon receiving (label, i, d, ε) from d do:
  P ← {t ; ε} ;
  send (path; P) to d;
结点 r(≠s, t)处算法:
/ * 结点 r 已标号为 (i_r, F_r, ε_r)，这个三元组表示在第 i_r 次迭代时，r 标号为 i_r，
        其父结点是F_r，吞吐量为ε_r * /
upon receiving (label, i, d, ε) message from node d do:
    if i = i_r       / * 在本次迭代时 r 已标号过 * /
          then send (echo, r) message to d   / * 拒绝标号请求 * /
          else if i > i_r      / * r在本次迭代时尚未标号 * /
                then F_r ← d ;
                    L_r ← L_r − {d} ;
                    (i_r, F_r, ε_r) ← (i, d, ε) ;
                    if L_r = φ
                          then send (echo, r) to d
                          else select v∈L_r ;
                              L_r ← L_r − {v} ;
                              send (label, i, r, α) to v,  α = min{ε,δ(r,v)}
                    endif
            endif
```

```
        endif;
    upon receiving (echo, d) from node d do:
        if L_r = φ
            then send(echo, r) message to father F_r
            else select v∈L_r;
                L_r ← L_r − {v};
                send(label, i_r, r, α) to v,  α = min{ε, δ(r,v)}
        endif;
    upon receiving (path; P) from node d do:
                P ← {r}∪P;
                δ(r, d) ← δ(r, d) − ε;
                if δ(r, d) = 0 then R_r ← R_r − {d} endif;
                δ(r, F_r) ← δ(r, F_r) + ε;
                R_r ← R_r ∪ {F_r};
                send (path; P) to F_r
    end.
```

定理 10.13 在基于异步通信的分布式计算模型上，计算一个有向网络 $N=(G,s,t,c)$ 的最大流，若 c 为 $E\rightarrow I^{+}$ 函数，且最大流值为 f^{*}，则算法 10.21 的通信复杂性为 $O(mf^{*})$。

证明 在深度优先搜索过程中，每次迭代的标号过程，终止于终点 t 被标号。在最坏情况下有两种情形出现：其一，仅在网络的其它所有顶点标号后才标号终点；其二，每个顶点 r(s,t 除外）仅在对所有邻居试图标号之后才标号成功。上述第二种情形需 $|E_r|$ 条标号消息和 $|E_r|-1$ 条应答消息，这里 $|E_r|$ 是 r 的非父结点的邻居数目。由于流量增广路径需 $|V|-1$ 条路径消息，故每次迭代需通信复杂性为 $\sum_{r\in V-\{s,t\}}(2|E_r|-1)+(|V|-1)=O(m)$，又若边的容量为整数，Edmonds 等人[8] 已证明，这需 f^{*} 次迭代。故使用深度优先搜索技术设计的最大流问题分布式算法的通信复杂性为 $O(mf^{*})$。

Cheung 还给出了使用最大增量 (Largest Augmentation) 搜索技术以及宽度优先搜索技术的最大流分布式算法，同上面的算法基本上是类似的，这里就不再介绍了。

10.6 小　结

本章重点介绍了网络流图问题——AOE 网及最大流的并行算法，主要内容包括：基于共享存贮模型，介绍了 AOE 网的存在性测试、拓扑排序以及求关键路径的并行算

法；基于超立方及洗牌互连网络模型，介绍了拓扑排序以及求关键路径的算法；基于异步通信的分布式计算模型，讨论了求关键路径的分布式算法．在最大流问题的并行算法中，给出了一个有效的实现算法．最后，在分布式计算模型上，利用深度优先搜索技术，介绍了一个求最大流的分布式算法．

最近，Goldberg 等人基于一种标号法给出了最大流的另外一个分布式算法，该算法的通信复杂性是 $O(n^2m)$，时间复杂性为 $O(n^2)$[9]．

关于最大流问题的并行算法，Johnson 曾基于 SIMD—CREW PRAM，给出了平面图的最大流算法．他的算法需 $O(\log^3 n)$ 时间、$O(n^4)$ 处理器或 $O(\log^2 n)$ 时间、$O(n^6)$ 处理器[10]。最近，Goldberg 等人基于 SIMD－EREW PRAM，对一般图给出了一个 $O(n^2\log n)$ 时间、$O(n)$ 处理器的最大流算法[9]。另外作者基于 SIMD－CREW PRAM，对 AOE 网的存在性测试，拓扑排序以及关键路径的计算，曾建议了更简单的并行算法[12]，且在时间复杂性不变的情况下处理器数目略有减少．

参 考 文 献

[1] Chaudhuri P, Ghosh R K. Parallel Algorithms for Analyzing Activity Networks, *BIT*, 26, 1986, 418–429

[2] Shaloach Y, Vishkin U. Finding the Maximum, Merging and Sorting in a Parallel Computation Model, *J. Algorithms*, 2, 1981, 88–102

[3] Dekel E, Nassimi D, Sahni S. Parallel Matrix and Graph Algorithms, *SIAM J. Comput.*, 10, 1981, 657–675

[4] Chaudhuri P. Distributed Algorithm for Analysing Networks, *Intern. J. Electronices*, 60(5), 1986, 603–607

[5] Shiloach Y, Vishkin U. An O(n^2log*n*) Parallel Max–flow Algorithm, *J.Algorithms*, 3, 1982, 128–146

[6] Brent R P. The Parallel Evaluation of General Arithmetic Expression, *J.ACM*, 21(2), 1974, 201–208

[7] Cheung T Y. Graph Traversal Techniques and the Maximum Flow Problem in Distributed Computation, *IEEE Trans. Software Enger.*, SE–9(4), 1983, 504–512

[8] Edmonds J, Karp R M. Theoretical Improvments in Algorithmic Efficiency for Network Flow Problems, *J.ACM*, 19(2), 1972, 248–264

[9] Goldberg A V, Tarjan R E. A New Approach to the Maximum–Flow Problem, *J.ACM*, 35(4), 1988, 921–940

[10] Johnson D B. Parallel Algorithms for Minimum Cuts and Maximum Flows in Planar Networks, *J. ACM*, 34(4), 1987, 950–967

[11] Nassimi D, Sahni S. Data Broadcasting in SIMD Computers, *IEEE Trans. Comput.*, C–30(2), 1981, 101–107

[12] 唐策善，梁维发，AOE网的并行算法，计算数学，13(2), 1991, 113–120

第十一章　极大独立集的并行算法

11.1 引　　言

近几年来，图的极大独立集问题的并行算法研究，已经取得许多令人鼓舞的成果．人们之所以热衷于这一问题的并行算法的研究，至少有两方面的原因：其一是它有广泛的应用背景，图的极大独立集问题可归约为最大约束装箱、最大匹配以及图的最大着色等调度问题[1,2]；其二是一般图的极大独立集问题的求解算法本质上是顺序的，若将其移植修改成并行算法，则效率很低[4]，因而从这一问题本身出发，探索其并行性是刺激人们对其进行研究的另一动机。

在本章中，我们将介绍几个极大独立集问题的并行算法．首先将叙述简单的并行算法和基本概念，然后再讨论有效的并行算法．因为有的算法引进了随机性，所以下一节讨论概率算法的基本知识[13,14]．

11.2 概率算法的基本概念

在分析算法时，通常分别考虑算法的最坏情况复杂性和平均情况复杂性。这对研究算法的性能，对许多问题是合适的，但也有不少问题不尽人意．尤其是有的问题个别实例求解需要很长的计算时间，而求解此问题的平均时间（又称期望时间），与最坏时间相比，却要小得多，在这种情况下，两者之间的差别就相当悬殊．从实用角度出发，如果要求解的一个问题包含有众多的实例，那么最坏情况的时间复杂性就难以测度算法的优劣，而平均时间却能成为衡量算法好坏的一个比较合理的标准。为此，人们就试图研究出一些平均时间复杂性较好的算法．但是，要分析一个算法的平均复杂性，需对输入数据的分布做出假设．如排序 n 个数的问题，一般假定输入的数据是均匀分布的，即 n 个数据中的每一个，在 n 个位置上出现的概率是等同的．然而遗憾的是：有许多计算问题的数据分布，要么是无法预先知道，要么是随时间的变化而变化，因而也就无法对其分布作出合理的假设。为了避免这种困难，人们试图对输入数据的概率分布不作任何假设，完全任其"自然"地分布，但在算法的内部引入随机性，从而可望得到一个平均时间较短、较实用的算法，这就是所谓的概率算法，或称随机算法．

假设 P 是一个待计算的问题，如果 P 中的每个计算任务称为 P 的一个实例，那么 P 本身就是这些计算任务的总和。例如，排序问题，就是汇集了所有的 n 元组 $I=(x_1, x_2, \cdots, x_n)$，即全体实例．对于一个给定实例 I，其任务就是产生某个置换 π，π 是 $\{1, 2, \cdots, n\}$ 的一个枚举，使得 $x_{\pi(1)} \leqslant x_{\pi(2)} \leqslant \cdots \leqslant x_{\pi(n)}$，即将实例 I 排成了非降次序．

如果我们要研究某个问题 P 的复杂性，那么总要涉及到它的实例 $I\in P$ 的规模 $|I|$。通常，将 $|I|$ 与其所需的关键操作多少、或所占用的存贮空间多少、或所需的消息量大小联系在一起。因此，对于一个 n 元组 $(x_1, x_2, \cdots, x_n)$，一般就假定其规模为 n。一个具有 n 个顶点、m 条边的（有向或无向）图 G，它的规模 $|G|$ 定义为 $n+m$。一个整数 n 的规模 $|n|$ 定义为二进制表示 n 时所需的位数，即 $|n|=\lceil \log n \rceil$。

设 P 表示一个计算问题，P 的一个概率算法 (Probability Algorithm) PA 定义为：令 I 是 P 的一个实例，$I\in P$，$|I|=n$，在用 PA 解决 I 的过程中，在它的某些确定的步骤上，随机地选取一个整数 b，$1\leqslant b<n$，用 b_i 表示选取的第 i 个元素，用 $r=(b_1, \cdots, b_s)$ 表示整个过程中所选出的随机序列。算法 PA 除了这些选取步骤外，其它步骤都是完全确定的。

若对每个 $I\in P$，$|I|=n$，算法 PA 解决 I 所需的平均时间小于或等于 $f(n)$，则称 PA 在期望时间 $f(n)$ 内解决问题 P。所谓**期望时间**，就是用 PA 解决所有 I 所用时间的平均数，该平均数是对所有可能的选择序列 r 而言的，且所有的序列 r 有相同的概率。

关于概率算法，存在几种不同理解。事实上，在上面叙述中，我们称 PA 在期望时间 $f(n)$ 内解决问题 P，并不意味着对实例 $I\in P$，$|I|=n$，PA 均能在时间 $f(n)$ 内结束，或者说在时间 $f(n)$ 内结束且给出正确的结果。换句话说，PA 在最坏情况下的运行时间，可能远远大于 $f(n)$，只不过发生这种情况的概率很小。因此导致对概率算法的两种不同理解方式：

第一种观点认为：PA 偶尔产生一个不正确的解，但总能在期望的时间内结束。设 PA 为某一概率算法，它对于 P 的某些实例 I 及其选择 r，可能产生一个不正确的解。令 $0<\varepsilon<1$。如果对任何一个 $I\in P$，PA 在期望时间产生一个不正确解的概率小于 ε，那么我们称 PA 解决问题 P 的概率大于 $1-\varepsilon$。换句话说，对于实例 I，导致 PA 产生一个不正确解的选择序列 r 的概率小于 ε。

第二种观点认为：PA 总能得到正确的解只不过其运行时间可能很大，甚至无穷。我们可以对第一种观点作适当的修改，即允许选择序列 r 的长度可以是无穷的。如果 r 的长度超过 m 的概率为 $P(I,m)$，且当 m 变大时，$P(I,m)$ 尽快的趋向于 0，那么用算法 PA 计算实例 I 最终能结束的概率为 1，且仍具有短的期望时间。

针对上面两种不同的观点，为了保持叙述的准确性，在提到算法 PA 解决某一问题 P 时，我们总称 PA 能在 $f(n)$ 时间内结束且得出正确解的概率大于 $1-\varepsilon$，这里 ε 是满足 $0\leqslant\varepsilon<1$ 的一个任意小的正数。

实际上，求解一个问题所用的时间方差也具有很重要的意义。因为它能度量随机变量与其平均值的偏离程度。

在以后的讨论中，约定 $P(E_0)$ 表示随机事件 E_0 发生的概率；$E(\xi)$ 表示随机变量 ξ 的数学期望；$\sigma(\xi)$ 表示随机变量 ξ 的方差，且方差和数学期望的关系是：

$$\sigma(\xi)=E(\xi-E(\xi))^2$$

11.3 极大独立集问题的简单并行算法

11.3.1 极大独立集问题的基本知识

设 $G(V,E)$ 是一个无向图，对任意一个顶点子集 $S \subseteq V$，令 $N(S)$ 是 S 的邻居集，定义 $N(S)=\{w \mid$ 存在一个 $u \in S$ 且 $(u,w) \in E\}$。若 $S \cap N(S)=\phi$，则 S 是 G 的一个独立集 (Independent Set) 也就是说，在 S 中任何两个顶点都没有边关联。若 S 不含任何这样一个真子集 S'，且 S' 也是一个独立集，则独立集 S 称为**极大独立集** (Maximal Independent Set，简记为 MIS)。换句话说，若 $S \cap N(S)=\phi$ 且 $S \cup N(S)=V$，则 $S \subseteq V$ 是极大独立集。在图 11.1 中，图 (a) 是一个独立集的例子，$S=\{1\}$，$N(S)=\{2,3,4,5,6\}$；图 (b) 是一个极大独立集的例子，$S=\{3,6\}$，$N(S)=\{1,2,4,5\}$。值得注意的是：极大独立集可能不唯一。例如图 11.1(b) 中，除了 $\{3,6\}$ 是极大独立集外，$\{3,5\}$，$\{2,4\}$，$\{2,5\}$，$\{4,6\}$ 都是该图的极大独立集。

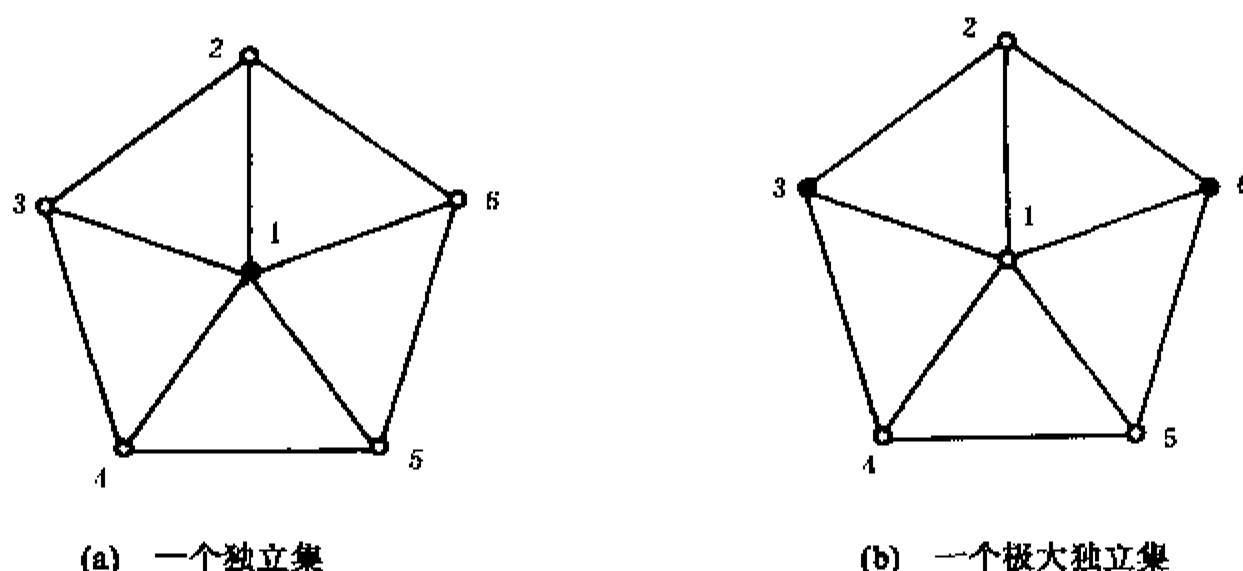


图 11.1 独立集和极大独立集示例

设 $G(V,E)$ 是一个无向图，$V=\{1,2,\cdots,n\}$，计算 G 的极大独立集 I 的串行算法非常简单，描述如下：

算法 11.1 FINDING MIS (SISD)

输入：图 $G(V,E)$ 的邻接矩阵 A；
输出：G 的一个极大独立集 $I \subseteq V$。

```
begin
(1) I←φ;    N(I)←φ;        /* 置极大独立集和邻集为空 */
(2) for i←1 to n do
        if i∉N(I) then I←I∪{i};
                  N(I)←N(I)∪{j |A(i,j)=1}
        endif
    endfor
end .
```

这个算法输出的极大独立集 I 称为字典序优先的极大独立集 (Lexicographically First Maximal Independent Set)，记作 LFMIS，显然它的时间复杂性为 $O(n)$。

尽管解决 MIS 问题的串行算法非常简单，但求解这个问题并行算法却难得多．Valiant[3] 曾注意到 MIS 问题很可能是那一类不存在快速并行算法的问题．后来 Cook 证明：在并行计算模型上，不存在一个时间复杂性为 $O(\log^{c_1} n)$、处理器数为 $O(n^{c_2})$ 的 LFMIS 并行算法[4]，这里 c_1 和 c_2 是大于 0 的常数．

因此，为了设计一个 MIS 问题的并行算法，需要采用同串行算法完全不同的设计策略．Karp 等人发展了一种称为迭代改进的设计策略，首次给出一个快速地求解 MIS 问题的并行算法。下面我们介绍他们算法的梗概．

11.3.2 极大独立集问题并行算法的高层描述

目前几乎所有的 MIS 并行算法都遵循 Karp 等人描述的框架结构．设 $G(V,E)$ 是一个无向图．令 $G'=(V',E')$ 是 G 的一个子图．对所有的 $W\subseteq V'$，我们定义 W 在 G' 中的邻居集 $N(W)$ 为：

$$N(W)=\{i \mid i\in V'\text{且存在}j\text{满足}(i,j)\in E'\}$$

算法描述如下：

算法 11.2 FINDING MIS IN PARALLEL

输入：无向图 $G(V,E)$；

输出：G 的一个极大独立集 I，$I\subseteq V$。

begin

(1) $I\leftarrow\phi$; $G'(V',E')\leftarrow G(V,E)$;

(2) **while** $G'\neq\phi$ **do**

(2.1) select a set $I'\subseteq V'$ which is independent in G';

(2.2) $I\leftarrow I\cup I'$;

(2.3) $Y\leftarrow I'\cup N(I')$;

(2.4) $G'=(V',E')$ is the induced subgraph on $V'-Y$

endwhile

end.

在算法 11.2 的执行过程中，新的子图 G' 的顶点集 V'，和集合 I、$N(I)$ 三者之间的关系，满足条件：

$$I\cap N(I)\cap V'=\phi \quad \text{和} \quad I\cup N(I)\cup V'=V$$

当算法终止时，有 $G'=\phi$，按照极大独立集的定义，这时候的 I 即是 G 的极大独立集．算法 11.2 的关键在第 (2.1) 步，若使用 $O(m)$ 处理器，一旦选出了 I'，则 I 及 G' 很容易在 $O(\log^2 n)$ 时间内修改完毕．还可证明：在 SIMD－EREW PRAM 上，在期望时间 $(\log^2 n)$ 内可选出 I，且

$$|I'\cup N(I')|=\Omega(|V'|/\log|V'|)$$

定理 11.1 在 SIMD－EREW PRAM 上，计算 MIS 问题的随机并行算法 11.2，需期

望 $O(\log^4 n)$ 时间、$O(n^2)$ 处理器；而它的确定算法版本需 $O(\log^4 n)$ 时间、$O(n^3/\log^3 n)$ 处理器。

证明 见Karp等人文章[1]。

鉴于 Karp 算法的复杂性分析非常复杂，这里不打算进一步详细地介绍他们的算法。下面先介绍 Luby 建议的两个简单的随机并行算法[2]，然后叙述把一个随机算法转换成一个确定算法的基本技术，并给出一个确定的 MIS 并行算法。

11.3.3 极大独立集问题的随机并行算法

Luby 建议的两个简单的 MIS 随机并行算法[2,5]，也遵循上面描述的框架结构。严格地讲，各种不同的 MIS 并行算法的主要区别，仅仅体现在选择步第 (2.1) 步的设计上。设计选择步必须满足下述两条性质:

1) 在并行计算模型上，选择步的执行时间应当非常短;

2) 在G'成为空集之前，整个while循环体的执行次数，应该非常的少。

首先我们介绍一个极其简单的 MIS 随机并行算法。不失一般性，假定 $V'=\{1, 2, \cdots, n'\}$。对所有 $i\in V'$，我们定义 $\mathrm{adj}(i)=\{j \mid j\in V'$ 且 $(i,j)\in E'\}$。

算法11.3 SIMPLE RANDOMIZED PARALLEL ALGORITHM FOR MIS

输入：无向图$G(V,E)$；

输出：G的一个极大独立集I，$I\subseteq V$。

begin

(1) $I\leftarrow\phi$; $G'(V',E')\leftarrow G(V,E)$;

(2) **while** $G'\neq\phi$ **do** / * (2.1)~(2.3)为选择步 * /

(2.1) **for each** i: $i\in V'$ **pardo** / * 为V'选择一个随机定序（枚举）π * /

$\pi(i)\leftarrow$ a number randomly chosen from $\{1,\cdots,n^4\}$

endfor ;

(2.2) $I'\leftarrow V'$;

(2.3) **for each** i, j: $(i,j)\in E'$ **pardo**

if $\pi(i)\geqslant\pi(j)$ **then** $I'\leftarrow I'-\{i\}$ **else** $I'\leftarrow I'-\{j\}$ **endif**

endfor ;

(2.4) $I\leftarrow I\cup I'$;

(2.5) $Y\leftarrow I'\cup N(I')$;

(2.6) $G'=(V',E')$ is the induced subgraph on $V'-Y$

endwhile

end .

在算法 11.3 中，所有随机变量 $\pi(i)$, $i\in V'$ 值的选取是相互独立的。这似乎是超出了 MIS 算法框架结构的思想，因为在某些情况下，完全有可能存在这样的边 $(i,j)\in E'$

且 $\pi(i)=\pi(j)$。然而，这并不影响我们的讨论，由于任何一对有边关联的顶点 i 和 j，它们具有相同 π 值的概率仅为 $1/n^4$。而顶点集 V' 中至多有 C_2^n 个顶点对，故 π 是 V' 中的一个随机定序的概率至少为 $1-1/2n^2$。

现在我们介绍另一个较算法 11.3 复杂一点的随机并行算法，它将在后面被转换成确定的并行算法。

令 $d(i)$ 是顶点 $i\in V'$ 在 $G'(V',E')$ 中的度数，集合 $\Psi=\{\text{coin}(i)|i\in V'\}$ 是相互独立的 0／1 随机变量集合。若 $d(i)\geqslant 1$，则 coin(i) 等于 1 的概率是 $1/2d(i)$；若 $d(i)=0$，则 coin(i) 恒为 1。另一个计算 MIS 的随机并行算法描述如下:

算法11.4 COMPLEXED RANDOMIZED MIS ALGORITHM

```
输入: 无向图G(V,E);
输出: G的一个极大独立集I, I ⊆ V.
begin
(1)  I ← φ;    G'(V',E') ← G(V,E);
(2)  while G' ≠ φ do
    (2.1)  for each i: i∈V' pardo
               d(i) ← |{j |(i,j)∈E'}|
           endfor;
    (2.2)  I' ← φ;
               /* 选择步(2.3)~(2.4) */
    (2.3)  for each i: i∈V' pardo
         (2.3.1)  if d(i) = 0
                      then coin(i) ← 1
                      else coin(i) ← randomly choose a value from {0,1}
                  endif;
         (2.3.2)  if coin(i) = 1 then  I' ← I'∪{i} endif
           endfor;
    (2.4)  for each i,j: (i,j)∈E' pardo
               if (i∈I') and (j∈I')
                   then if d(i) ≤ d(j)  then  I' ← I' − {i}
                                        else  I' ← I' − {j}
                        endif
               endif
           endfor;
    (2.5)  I ← I∪I';
    (2.6)  Y ← I'∪N(I');
    (2.7)  G'(V',E') is the induced subgraph on V' − Y
   endwhile
```

end.

11.3.4 随机并行算法的复杂性分析

现在我们来分析上述两个简单的随机并行算法的计算复杂性，它们的分析证明非常相似。在算法 11.3 和 11.4 中，**while** 循环的条件是 $G' \neq \phi$，当 $G' \neq \phi$ 时执行循环体。令 t_A 和 t_B 分别表示算法 11.3 和 11.4 在整个 **while** 循环体的执行次数。下面将证明：t_A 和 t_B 的期望执行次数都是 $O(\log n)$。我们将证明：在平均情况下，**while** 循环体的每次执行，都将使边集合 E' 中的边数减少一个常量因子。对算法 11.3 和 11.4 的 **while** 循环体来讲，若在第 k 次执行前 E' 中的边数，分别用 Y_k^A 和 Y_k^B 来表示，则第 k 次执行后删除 E' 中的边数分别为 $Y_k^A - Y_{k+1}^A$ 和 $Y_k^B - Y_{k+1}^B$。

对所有 $i \in V'$ 且 $d(i) \geqslant 1$，令

$$\mathrm{sum}(i) = \sum_{j \in \mathrm{adj}(i)} 1/d(j)$$

引理 11.1 令 $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n \geqslant 0$ 为实数变量，对任意 $l, 1 \leqslant l \leqslant n$，令

$$\alpha_l = \sum_{j=1}^{l} p_j, \qquad \beta_l = \sum_{j=1}^{l} \sum_{k=j+1}^{l} p_j p_k, \qquad \gamma_l = \alpha_l - c\beta_l$$

其中：c是大于0的常量，那么有

$$\max\{\gamma_l \mid 1 \leqslant l \leqslant n\} \geqslant \min\{\alpha_n,\ 1/c\}/2$$

证明 用归纳法证明。要使 β_l 极大化，只有 $p_1 = p_2 = \cdots = p_n = \alpha_l / l$，这就得到

$$\beta_l \leqslant \alpha_l^2 \cdot (l-1)/2l$$

因此

$$\gamma_l \geqslant \alpha_l(1 - c\alpha_l(l-1)/2l)$$

若 $\alpha_n \leqslant /c$，则 $\gamma_n \geqslant \alpha_n / 2$。若 $\alpha_1 \geqslant 1/c$，则 $\gamma_1 \geqslant 1/c$。否则，存在一个 l，$1 < l < n$ 且 $\alpha_{l-1} \leqslant 1/c \leqslant \alpha_l \leqslant 1/c \cdot l/(l-1)$。因为 $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$，所以不等式成立。因此我们有 $\gamma_l \geqslant 1/2c$。

引理 11.2 在算法 11.3 中，对图 G' 的任意一个顶点 $i \in V'$ 且 $d(i) \geqslant 1$，有

$$P(i \in N(I')) \geqslant \min\{\mathrm{sum}(i),\ 1\} \cdot (1 - 1/2n^2)/4$$

证明 设 π 是 V' 的一个随机定序，且概率至少是 $1 - 1/2n^2$。对任意的 $j \in V'$，定义 E_j 是 $\pi(j) < \min\{\pi(k) \mid k \in \mathrm{adj}(j)\}$ 的事件，令 $\mathrm{adj}(i) = \{1,2,\cdots,d(i)\}$，$p_j = P(E_j) = 1/(d(j)+1)$ 和 $p_1 \geqslant \cdots \geqslant p_{d(i)}$。那么，根据包含排斥原理，对所有的 l，$1 \leqslant l \leqslant d(i)$，有

$$P(i \in N(I')) \geqslant P(\bigcup_{j=1}^{l} E_j) \geqslant \sum_{j=1}^{l} p_j - \sum_{j=1}^{l} \sum_{k=j+1}^{l} P(E_j \cap E_k)$$

对固定的 j,k，$1 \leqslant j < k \leqslant l$，令 E' 是 $\pi(j) < \min\{\pi(v) \mid v \in \mathrm{adj}(j) \cup \mathrm{adj}(k)\}$ 的事件，令 E'_k 是 $\pi(k) < \min\{\pi(v) \mid v \in \mathrm{adj}(j) \cup \mathrm{adj}(k)\}$ 的事件，令 $d(j,k) = |\mathrm{adj}(j) \cup \mathrm{adj}(k)|$，那么，

$$P(E_j \cap E_k) \leqslant P(E'_j)P(E_k|E'_j) + P(E_j|E'_k)$$
$$\leqslant \frac{1}{d(j,k)+1}\left(\frac{1}{d(k)+1} + \frac{1}{d(j)+1}\right)$$
$$\leqslant 2p_j p_k$$

令 $\alpha = \sum_{j=1}^{d(i)} p_j$，由引理 11.1 可知，当 π 是 V' 的一个随机定序时，有

$$P(i \in N(I')) \geqslant \min\{\alpha,\ 1/2\}/2 \geqslant \min\{\mathrm{sum}(i),1\}/4$$

引理 11.3　在算法 11.4 中，对图 G' 的任意一个顶点 $i \in V'$，且 $d(i) \geqslant 1$，有

$$P(i \in N(I')) \geqslant \min\{\mathrm{sum}(i)/2,\ 1\}/4$$

证明　对任意一个 $j \in V'$，令 E_j 是 $\mathrm{coin}(j)=1$ 的事件，且令 $p_j = P(E_j) = 1/2d(j)$ ，不失一般性，令 $\mathrm{adj}(i) = \{1,\cdots,d(i)\}$，$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_{d(i)}$　令 E'_1 是事件 E_1，对 $2 \leqslant j \leqslant d(i)$，定义事件 E'_j 如下:

$$E'_j = (\bigcap_{k=1}^{j-1} \sim E_k) \cap E_j, \qquad A_j = \bigcap_{\substack{v \in \mathrm{adj}(j) \\ d(v) \geqslant d(j)}} \sim E_v$$

那么

$$P(i \in N(I')) \geqslant \sum_{j=1}^{d(i)} P(E'_j)P(A_j|E'_j)$$

但

$$P(A_j|E'_j) \geqslant P(A_j) \geqslant 1 - \sum_{\substack{v \in \mathrm{adj}(j) \\ d(v) \geqslant d(j)}} p \geqslant 1/2$$

且

$$\sum_{j=1}^{d(i)} P(E'_j) = P(\bigcup_{j=1}^{d(i)} E_j)$$

对 $k \neq j$ 来讲，$P(E_j \cap E_k) = p_j \cdot p_k$．因此由包含排斥原理，对 $1 \leqslant l \leqslant d(j)$ 有

$$P(\bigcup_{j=1}^{d(i)} E_j) \geqslant P(\bigcup_{j=1}^{l} E_j) \geqslant \sum_{j=1}^{l} p_j - \sum_{j=1}^{l}\sum_{k=j+1}^{l} p_j \cdot p_k$$

令 $\alpha = \sum_{j=1}^{d(i)} p_j$，由引理 11.1 可知，

$$P(\bigcup_{j=1}^{d(i)} E_j) \geqslant \min\{\alpha,1\}/2 \geqslant \min\{\mathrm{sum}(i)/2,\ 1\}/2$$

故推得

$$P(i \in N(I')) \geqslant \min\{\mathrm{sum}(i)/2,\ 1\}/4$$

定理 11.2　在 SIMD－CRCW PRAM 上，并行随机算法 11.3 及 11.4 中的 while 循环体，第 k 次执行时，删除边的期望值分别为

(1)　$E(Y_k^A - Y_{k+1}^A) \geqslant Y_k^A/8 - 1/16$

(2) $E(Y_k^B - Y_{k+1}^B) \geqslant Y_k^B / 8$

证明 设在第 k 次执行 while 循环体前的图是 $G'(V',E')$。在 while 循环体执行中删除的边，其端点至少有一个是在集合 $I' \cup N(I')$ 中，即对每条被删除的边 (i,j)，或者 $i \in I' \cup (I')$，或者 $j \in I' \cup N(I)$。因此，

$$E(Y_k^B - Y_{k+1}^B) \geqslant \sum_{i \in V'} d(i)\, P(i \in I' \cup N(I'))/2$$

$$\geqslant \sum_{i \in V'} d(i)\, P(i \in N(I'))/2$$

定理剩下的证明，分别要引用引理 11.2 和引理 11.3，这里我们只证明后者.

由引理11.3可知，

$$P(i \in N(I') \geqslant \min\{\mathrm{sum}(i)/2,\ 1\}/4$$

因此，

$$E(Y_k^B - Y_{k+1}^B) \geqslant \left(\sum_{\substack{i \in V' \\ \mathrm{sum}(i) \leqslant 2}} d(i)\mathrm{sum}(i)/2 + \sum_{\substack{i \in V' \\ \mathrm{sum}(i) > 2}} d(i) \right) / 8$$

$$\geqslant \left(\sum_{\substack{i \in V' \\ \mathrm{sum}(i) \leqslant 2}} \sum_{j \in \mathrm{adj}(i)} d(i)/2d(j) + \sum_{\substack{i \in V' \\ \mathrm{sum}(i) > 2}} \sum_{j \in \mathrm{adj}(i)} 1 \right) / 8$$

$$\geqslant \left(\sum_{\substack{(i,j) \in E' \\ \mathrm{sum}(i) \leqslant 2 \\ \mathrm{sum}(j) \leqslant 2}} (d(i)/d(j) + d(j)/d(i)) \right) / 2 + \sum_{\substack{(i,j) \in E' \\ \mathrm{sum}(i) \leqslant 2 \\ \mathrm{sum}(j) > 2}} (d(i)/2d(j) + 1)$$

$$+ \sum_{\substack{(i,j) \in E' \\ \mathrm{sum}(i) > 2 \\ \mathrm{sum}(j) > 2}} 2/8$$

$$\geqslant |E'|/8 = Y_k^B/8$$

类似地可以证得$E(Y_k^A - Y_{k+1}^A) \geqslant Y_k^A/8 - 1/16$.

定理 11.3 在 SIMD－CRCW PRAM 上，计算一个无向图 $G(V,E)$，$|V| = n$，$|E| = m$ 的极大独立集的随机并行算法 11.3，它的期望时间为 $O(\log n)$、处理器数为 $O(m)$.

证明 由定理 11.2 我们知道，在 SIMD－CRCW PRAM 上，每次执行 while 循环体时，从 E' 中删除的边数，其期望值为大于 0 的一个常数．故到 $G' = \phi$ 时，while 循环体期望的执行次数是 $O(\log n)$。而算法使用了 m 个处理器 $(m \geqslant n)$，其余各步均可在 $O(1)$ 时间内完成。所以算法 11.3 的期望执行时间为 $O(\log n)$、处理器数为 $O(m)$.

定理 11.4 在 SIMD－EREW PRAM 上，计算一个无向图 $G(V,E)$，$|V| = n$，$|E| = m$ 的极大独立集的随机并行算法 11.4，其期望运行时间为 $O(\log^2 n)$、处理器数为 $O(m)$。

证明 由定理 11.3 知，算法 11.4 的 while 循环体期望的执行次数为 $O(\log n)$，而在 SIMD－EREW PRAM 上实现集合归并等运算需 $O(\log n)$ 时间、$O(m)$ 处理器．故整个算法的期望执行时间为 $O(\log^2 n)$、处理器数为 $O(m)$，$m \geqslant cn$，c 为大于 0 的一个常数．

11.3.5 极大独立集问题的并行确定性算法

通常，把算法中每步执行是确定的、而不具有随机选择的一类算法称之为确定性算法。那么，什么样的随机算法可以转换成等价的确定性算法呢？它们之间是怎样转换的？现在来研究这些问题。首先我们给出转换的基本策略，然后利用这种策略，将随机并行算法 11.4 转换成一个确定性并行算法。

1. **转换的基本策略**

要将一个随机并行算法转换成一个确定性并行算法，为了转换方便，应尽量这样来描述随机算法：将算法的随机性都包括在一个选择步里，这一步在算法执行过程中可能多次执行；在选择步里，随机变量 $X_0, X_1, \cdots, X_{n-1}$ 的值的选择是互相独立的，因而从平均角度来讲，这些随机变量的值集合是好的 (Good)；算法在执行选择步时能很快地判别一组随机变量的值集合的好坏，且迅速地选择好的值执行下去。算法分析证明：若每次执行选择步都选择了一个好的值集合，则在一个给定的时间界内，算法输出一个正确的解。除此之外，还应附加一些限制条件，即是：

(1) 设 $r>0$ 是一个整数，$q \geqslant n$ 是一个素数，且 r 和 q 囿界于 n 的多项式内，则随机变量 X_0, X_1, $\cdots$, X_{n-1} 中的 X_i 取值范围是 $R=\{R_1, \cdots, R_r\}$，而且 X_i 的值为 R_j 的概率是 n_{ij}/q，其中 n_{ij} 是一个大于 0 的整数，且 $q=\sum_{j=1}^{r} n_{ij}$；

(2) 在算法分析时应当证明：若 X_0，X_1，$\cdots$，X_{n-1} 仅仅是成对独立的 (Pairwise Independent) 的随机变量，则随机变量 X_0，X_1，$\cdots$，X_{n-1} 的值集合是好的值集合的概率大于 0，即好的值集合总是存在的。

2. **选择步的模拟实现**

在随机并行算法 11.4 中，除了选择步引入了随机性以外，其它步骤都是确定的，毋需进行确定性模拟。因此，我们只要对选择步并行地进行确定性模拟。选择步的模拟如下：

构造一个具有 q^2 个样本点的概率空间，在这个空间中，每个样本点对应 X_0，$X_1, \cdots, X_{n-1}$ 的一个值集合。并行地产生 q^2 个算法的拷贝，每个样本点赋给一个拷贝，并行地测试哪些样本点是好的，选取对应的 $X_0, X_1, \cdots, X_{n-1}$ 的值集合是好的样本点作为选择步的输出。

由于具有大于 0 概率的随机样本点是好的，故至少存在一个样本点一定是好的。又因为在每次执行选择步时都采用 $X_0, X_1, \cdots$，X_{n-1} 的一个好的值集合，所以转换后的算法是确定的，并保证在给定的时间界内执行完毕。

令 $X_0, X_1, \cdots, X_{n-1}$ 是满足约束条件 (1) 的随机变量。下面将证明：在生成的 q^2 个样本点的概率空间中，随机变量是成对独立的。在一个随机变量相互独立的概率空间中，若每个随机变量具有非零概率值至少有两个，则得到的样本点的数目至少是 $\Omega(2^n)$。

怎样产生上述性质的概率空间呢？我们现在介绍如下：考虑一个 $n \times q$ 的矩阵 A，A 的第 i 行对应随机变量 X_i 的可能取值，则 A 的第 i 行恰好含有 n_{ij} 项等于 R_j。令 0

$\leqslant x,\ y \leqslant q-1$，每个样本点 $b^{(x,y)}$ 定义为:

$$b^{(x,y)} = (b_0,\ b_1,\ \cdots,\ b_{n-1})$$

其中 $b_i = A_{i,(x+y\cdot i)\bmod q}$ 是 X_i 在样本点 $b^{(x,y)}$ 的值．我们赋给每个样本点的概率为 $1/q^2$．则得到以下的诸引理．

引理 11.4　$P(X_i = R_j) = n_{ij}/q$.

证明　固定 l，恰好存在 q 对 x,y，它们满足: $(x+y\cdot i)\bmod q = l$. 有 l 个 n_{ij} 值且 $A_{i,l} = R_j$.

引理 11.5　$P(X_i = R_j \text{ and } X_{i'} = R_{j'}) = (n_{ij}\cdot n_{i'j'})/q^2$.

证明　若固定 l 及 l'，恰好存在一对 x,y，使得 $(x+y\cdot i)\bmod q = l$ 及 $(x+y\cdot i')\bmod q = l'$ 同时成立．

引理 11.5 证明了随机变量 $X_0, X_1, \cdots, X_{n-1}$ 在构造出的概率空间中是成对独立的．Luby 还把成对独立的概念推广到 $d-$ 对独立概念上，并给出了一般的产生 $d-$ 对独立随机变量的概率空间的方法[2]，这里就不再介绍了．

在算法 11.4 中，若随机变量 coin(i), $i\in V'$ 仅仅是成对独立的，则对任意 $i\in V'$，定义 E_i 是 coin(i) = 1 事件，令 $p_i = P(E_i) = 1/2d(i)$．下面我们将证明：在新的定义下，算法 11.4 有关的引理及定理仍然成立．

引理11.6　$P(i\in N(I')) \geqslant \min\{\mathrm{sum}(i),\ 1\}/8$.

证明　令 $\alpha_0 = 0$，对 $1 \leqslant l \leqslant d(i)$，令 $\alpha_l = \sum_{j=1}^{l} p_j$，则

$$P(i\in N(I')) \geqslant \sum_{j=1}^{d(i)} P(E'_j)P(A_j|E'_j)$$

又因为

$$P(A_j|E'_j) = 1 - P(\sim A_j|E'_j)$$

但是

$$P(\sim A_j|E_j') \leqslant \sum_{\substack{v\in \mathrm{adj}(j)\\ d(v)\geqslant d(j)}} P(E_v|E_j')$$

且

$$\begin{aligned} P(E_v|E_j') &= P(E_v \cap \sim E_1 \cap\cdots\cap \sim E_{j-1}|E_j)/P(\sim E_1\cap\cdots\cap\sim E_{j-1}|E_j) \\ &\leqslant P(E_v|E_j)/(1-P(\bigcup_{l=1}^{j-1}E_l|E_j)) \\ &\leqslant p_v/(1-\sum_{l=1}^{j-1}P(E_l|E_j)) \\ &= p_v/(1-\alpha_{j-1}) \end{aligned}$$

故

$$P(\sim A_j|E_j') \leqslant \sum_{\substack{v \in adj(j) \\ d(v) \geqslant d(j)}} P_v / (1-\alpha_{j-1}) \leqslant 1/2(1-\alpha_{j-1})$$

因此

$$P(A_j|E_j') = (1 - P(\sim A_j|E_j') \geqslant (1-2\alpha_{j-1})/2(1-\alpha_{j-1})$$

$$P(E'_j) = P(E_j)P(\sim E_1 \cap \cdots \cap \sim E_{j-1}|E_j)$$

$$= p_j(1 - P(\bigcup_{l=1}^{j-1} E_l|E_j)) \geqslant p_j(1-\alpha_{j-1})$$

所以，对 $1 \leqslant l \leqslant d(v)$ 且 $\alpha_l < 1/2$，有

$$P(i \in N(I')) \geqslant \sum_{j=1}^{l} p_j(1-2\alpha_{j-1})/2$$

$$= (\sum_{j=1}^{l} p_j - 2\sum_{j=1}^{l}\sum_{k=l+1}^{l} p_j \cdot p_k)/2$$

由引理11.1得

$$P(i \in N(i')) \geqslant \min\{\alpha_{d(i)}, 1/2\}/4$$

$$\geqslant \min\{\text{sum}(i), 1\}/8$$

定理 11.5　当随机变量 $\{\text{coin}(i)|i \in V'\}$ 是成对独立时，$E(Y_k^B - Y_{k+1}^B) \geqslant Y_k^B/16$。

证明　类似定理 11.2 的证明，只需在证明中用引理 11.5 替换引理 11.3 即可。

由引理 11.3、11.4 及定理 11.5 可知，算法 11.4 几乎满足了转换条件的要求。随机变量 $\{\text{coin}(i)|\ i \in V'\}$ 的取值范围是 $\{0,1\}$，若把它的一个值集合应用于选择步时，至少使 E' 的边数删除掉 1／16，则这个值集合是好的值集合。

定理 11.5 蕴含着：若随机变量 $\{\text{coin}(i)|\ i \in V'\}$ 是成对独立的，则在任何一个概率空间内至少有一个样本点对应的值集合是好的。

然而，根据随机变量 $\{\text{coin}(i)|\ i \in V'\}$ 的最初定义，它们不符合转换限制条件的第 (1) 条。为此，我们需对 $\{\text{coin}(i)|\ i \in V'\}$ 的定义做两点修改：对 $\{\text{coin}(i)|\ i \in V'\}$ 的赋值概率要修改为：令 q 是一个 $n \leqslant q \leqslant 2^n$ 的素数，我们赋予 $\text{coin}(i) = 1$ 的概率是 $p'_i = \lfloor p_i \cdot q \rfloor / q$，其中 $p_i = 1/2d(i)$；此外，对算法 11.4 也要作相应的修改，修改后的算法如下：

算法11.5　IMPROVED RANDOMIZED MIS ALGORITHM

输入：无向图$G(V,E)$；

输出：G的一个极大独立集I，$I \subseteq V$。

begin

(1)　$I \leftarrow \phi$；　$G'(V',E') \leftarrow G(V,E)$；

(2)　compute a prime q such that $n \leqslant q \leqslant 2n$；

(3)　**while** $G' \neq \phi$ **do**

　(3.1)　**for each** $i: i \in V'$ **pardo**　／＊计算每个顶点 i 的度数＊／

```
        d(i) ← |{j | (i,j)∈E'}|
      endfor ;
(3.2)  for each i: i∈V' pardo
            if d(i) = 0 then I ← I∪{i} ;  V' ← V' − {i}  endif
      endfor ;
(3.3)  find i∈V' such that d(i) = max{d(j) | j ∈V'} ;
(3.4)  if d(i) ≥ n / 16
        then
          (3.4.1)  I ← I∪{i} ;
          (3.4.2)  V' ← V' − {i}∪adj(i) ;
          (3.4.3)  E' ← E' − {(i,j) | j∈adj(i)}
        else  / *  ∀i∈V', d(i) < n / 16  * /
          (3.4.4)  randomly choose x and y such that 0 ≤ x,y ≤ q − 1 ;
          (3.4.5)  X ← φ ;
          (3.4.6)  for each i: i∈V' pardo
                     compute n(i) ← ⌊q / 2d(i)⌋ ;
                     compute l(i) ← (x + y * i) mod q;
                     if l(i) ≤ n(i) then X ← X∪{i} endif
                   endfor ;
          (3.4.7)  I' ← X ;
          (3.4.8)  for each i,j: (i,j)∈E'∧(i∈X)∧(j∈X) pardo
                         if d(i) ≤ d(j)  then I' ← I' − {i}
                                         else I' ← I − {j}
                         endif
                   endfor ;
          (3.4.9)  I ← I∪I' ;
          (3.4.10) Y ← I'∪N(I') ;
          (3.4.11) G'(V',E') is the induced subgraph on V' − Y
      endif
    endwhile
end .
```

在算法 11.5 中的第(3.4)步，满足 $d(i) \geq n/16$ 的情况至多出现 16 次。这是因为由定理 11.5 知，每次执行 while 循环体至少使原图的边数减少了 1/16，故在执行循环体 16 次以后，得到的图其顶点度数满足 $d(i) < n/16$，即$\forall i \in V'$，$d(i) < n/16$为真。它蕴含着$q/2d(i) \geq n/2d(i) > 8$。因而 $p'_i \geq 8q$，也就意味着 $\lfloor p_i q \rfloor / 8 \geq 1$。又因为$(\lfloor p_i q \rfloor + 1)/q \geq p_i$，所以$(8/9)p_i \leq p'_i \leq p_i$。

引理 11.7 设随机变量 $\{coin(i) \mid i \in V'\}$ 是成对独立的，且 $P(coin(i)=1)=p'_i$，$\forall i \in V'$，$d(i)<n/16$，那么 $P(i \in N(I')) \geqslant \min\{sum(i),1\}/9$.

证明 用 $8/9p_i$ 作为 $P(coin(i)=1$ 的下界，用 p_i 作为 $P(coin(i)=1)$ 的上界，除此之外，其余的证明与引理 11.6 相同。

定理 11.6 在算法 11.4 中，按修改后的随机变量 $\{coin(i) \mid i \in V'\}$ 赋值，且对任意 $i \in V'$，$d(i)<n/16$，则 $E(Y_k^B - Y_{k+1}^B) \geqslant Y_k^B/18$.

证明 类似定理 11.2 的证明。只要在证明过程中，用引理 11.7 替换引理 11.3 即可得证。

因此，若对任意 $i \in V'$ 均有 $d(i)<n/16$，则每次执行 **while** 循环体删除 E' 的边数，其期望值是整个图边数的 1/18。

定理 11.7 在 SIMD－EREW PRAM 上，计算一个无向图 $G(V,E)$，$|V|=n$，$|E|=m$ 的极大独立集，算法 11.5 的期望运行时间为 $O(\log^2 n)$、处理器数为 $O(m)$，$m \geqslant n$.

证明 在算法 11.5 中，第 (1) 步和第 (2) 步至多需 $O(1)$ 时间、$O(m)$ 处理器；由定理 11.6 知，**while** 循环体期望的执行次数为 $O(\log n)$。第 (3.1) 步需 $O(\log n)$ 时间、$O(m)$ 处理器；第 (3.2) 步需 $O(\log n)$ 时间、$O(n)$ 处理器；第 (3.3) 步需 $O(\log n)$ 时间、$O(n)$ 处理器；第 (3.4) 步中的每个子步至多需 $O(\log n)$ 时间、$O(m)$ 处理器。因此，算法 11.5 期望的运行时间为 $O(\log^2 n)$、处理器数为 $O(m)$。

由上面的叙述可知，算法 11.5 已满足了转换条件的所有要求，它的确定性转换就变得非常简单，只需并行测试对应随机变量 $\{coin(i) \mid i \in V'\}$ 值集合的概率空间中的 q^2 个样本点，然后从中选择这样一个值集合，使得该集合能从 G' 中删除的边数最多。故整个 **while** 循环体的执行次数至多需

$$(\log n^2 / \log\frac{18}{17}) + 16 \leqslant 25\log n + 16$$

3. 极大独立集问题确定性并行算法的形式化描述

有了上面的准备之后，现在我们描述极大独立集问题的确定性并行算法。

算法 11.6 DETERMINITIC PARALLEL ALGORITHM FOR MIS PROBLEM

```
输入：无向图G(V,E)；
输出：G的一个极大独立集I，I ⊆ V.
begin
(1)  I ← φ;   G'(V',E') ← G(V,E);
(2)  compute a prime q such that n ≤ q ≤ 2n ;
(3)  while G' ≠ φ do
   (3.1) for each i,j :(i,j )∈E' pardo
            d(i) ← |{j | (i,j )∈E'}|
         endfor ;
```

```
(3.2) for each i,x,y : i∈V', 1 ≤ x,y ≤ q pardo
                d(i,x,y) ← d(i)
      endfor ;
(3.3) for each i: i∈V' pardo
            if d(i) = 0 then I ← I∪{i} endif
      endfor ;
(3.4) for each j : j∈V' pardo
         d(i_0) ← max_j {d(j)| j∈V'}
      endfor ;
(3.5) if d(i_0) ≥ n / 16
          then
    (3.5.1)  I ← I∪{i_0} ;
    (3.5.2)  G' is a graph induced on the vertices V' − ({i_0}∪{adj {i_0})})
          else   / *  ∀i∈V', d(i) < n / 16   * /
    (3.5.3)  for each x,y : 0 ≤ x,y ≤ q − 1 pardo
    (3.5.4)     X(x,y) ← φ ;
    (3.5.5)     for each i: i∈V' pardo
                  n(i,x,y) ← ⌊q / 2d(i,x,y)⌋ ;
                  l(i,x,y) ← (x + y * i) mod q ;
                  if l(i,x,y) ≤ n(i,x,y)   then X(x,y) ← X(x,y)∪{i}   endif
                endfor ;
    (3.5.6)     I' ← X(x,y) ;
    (3.5.7)     for each i,j : i∈X(x,y)∧j∈X(x,y) pardo
                  if (i,j )∈E'
                      then if d(i,x,y) ≤ d(j ,x,y)
                               then I'(x,y) ← I'(x,y) − {i}
                               else I'(x,y) ← I'(x,y) − {j }
                           endif
                  endif
                endfor ;
    (3.5.8)     Y(x,y) ← I'(x,y)∪N(I'(x,y))
              endfor ;
    (3.5.9)   for each x,y: 1 ≤ x,y ≤ q pardo
                  find a set Y(x_0,y_0) such that
                          |Y(x_0,y_0)| = max_{x,y} {|Y(x,y)|1 ≤ x,y ≤ q}
              endfor ;
```

(3.5.10) $I \leftarrow I \cup I'(x_0, y_0)$;

(3.5.11) $G'(V', E')$ is the induced subgraph $V' - Y(x_0, y_0)$

endif

endwhile

end .

定理 11.8 在 SIMD－EREW PRAM 上，计算一个无向图 $G(V,E)$，$|V|=n$，$|E|=m$ 的极大独立集，确定性并行算法 11.6 需 $O(\log^2 n)$ 时间、$O(mn^2)$ 处理器．

证明 算法 11.5 与确定性算法 11.6 的区别在于：后者每次不是随机的选择一个样本点，而是对概率空间的每个样本点同时执行算法 11.5，因而所需的处理器数目为 $O(mq^2)$．又在 **while** 循环体中，与算法 11.5 不同的是：确定性算法 11.6 的第 (3.5.9) 步找最大值需 $O(\log q^2) = O(\log q)$ 时间，前面的分析已经得出，**while** 循环体至多执行 $25\log n + 16$ 次．所以确定性算法 11.6 需 $O(\log^2 n + \log n \log q) = O(\log^2 n)$ 时间、$O(mq^2) = O(mn^2)$ 处理器．

11.4 有效的极大独立集并行算法

11.4.1 基本概念

在本章引言中曾提到，无向图的 MIS 问题可归约为最大着色问题．通俗地讲，若在图中能找出它的 MIS，对 MIS 中的顶点着上同一种颜色，则不会和邻集顶点的颜色相同，可导致图的一个最大着色；反之，若在图的最大着色数的顶点集中，将着相同颜色的顶点汇集起来，它就是图的一个 MIS．在这一节中，我们通过对图的顶点着色来求图的 MIS．关于图的顶点着色，将在下一章进行详细的讨论．

在 SIMD－EREW PRAM 上，Goldberg 等人对 MIS 问题建议了第一个确定的并行算法[6]．本节就来介绍这个确定的并行算法．

设 $G(V,E)$ 是一个无向图，$V=\{0,1,\cdots,n-1\}$．令 $C \subseteq V$ 是一个顶点集合．$G[C]$ 表示 G 中由 C 内顶点导出的一个子图．G 的一个部分着色 Ψ 定义为：对 G 的部分顶点集 C_1，…，C_p 进行着色，且满足 $C_i \cap C_j = \phi$，$(i \neq j)$，$C_i \subseteq V$，$\bigcup_{i=1}^{p} C_i \subseteq V$．集合 C_i 中的顶点着第 i 种颜色 $(1 \leqslant i \leqslant p)$．若 $\bigcup_{i=1}^{p} C_i = V$，则称 Ψ 为 G 的一个完整着色，$\Psi=\{C_1, C_2, \cdots, C_p\}$．$G$ 的一个平凡着色 Ψ 指的是 $|\Psi|=|V|$，即是 G 的所有顶点都着上互不相同的颜色．为了后面的讨论方便，我们约定：没有着色的顶点置颜色标记为 0．已删除的顶点的着色标记为 -1．每步操作仅对着色标记 $\geqslant 0$ 的顶点进行．

设 $\Psi=\{C_1, C_2, \cdots, C_p\}$ 是一个部分着色集．定义矩阵 $D(\Psi)=(d_{ij})_{p\times p}$ 及函数 $Q(\Psi)$

如下:

$$d_{ij} = |N(C_i) \cap C_j|, \quad 1 \le i, j \le p;$$

$$Q(\Psi) = \max_{1 \le i \le p} \{|C_i| + |N(C_i)|\}$$

若 Ψ 是 G 的一个部分着色，且 $|\Psi| = p$，则由 Ψ 及常量 $h > 0$ 定义了一个无向图 $B_{\Psi,h}(V_1, E_1)$，其中 $V_1 = \{C_1, \cdots, C_p\}, E_1 = \{(C_i, C_j) \mid d_{ij} \ge h 且 d_{ji} \ge h\}$。图 $G(V,E)$ 的补图 $G^c(V,E')$ 定义为：$E' = (V \times V) - E$。

11.4.2 算法的形式化描述

我们采用自顶向下的策略进行描述，先用框架结构概略地描述整个算法，然后对算法的框架结构一步一步地展开，直至得出一个完整的详细算法为止。

1. MIS算法的框架结构

算法的描述框架也遵循 Karp 的结构，为了叙述的完整起见，这里再现这种框架如下:

算法11.7 MIS ALGORITHM BY KARP

输入：无向图 $G(V,E)$；

输出：G 的一个极大独立集 I，$I \subseteq V$。

begin

(1) $I \leftarrow \phi$;   / * I 是 G 的极大独立集 * /

(2) $A \leftarrow V$;

(3) **while** $A \ne \phi$ **do**

    (3.1) $C \leftarrow$ FINDSET(A);

    (3.2) $I \leftarrow I \cup C$;

    (3.3) $A \leftarrow A - (C \cup N(C))$

    endwhile

end.

若每次执行函数 FINDSET 仅需 $O(\log^{s_1} n)$ 时间，且产生一个独立集 C 满足 $|C \cup N(C)| = \Omega(|A| / \log^{s_2} |A|)$，则算法 11.7 可以在 $O(\log^{s_3} n)$ 时间内完成，其中 $s_1 > 0$, $s_2 > 0$, $s_3 \ge \max\{s_1, s_2\}$，且它们均为常量。

2. 函数 FINDSET

函数 FINDSET 的目标试图从 $H = G[A]$ 中找出一个独立集 C，使得 C 满足 $|C \cup N(C)| \ge c_0 k / \log k$，$k = |A|$，$c_0$ 为一个正常数。为此，FINDSET 实现如下：开始时对 $G[A]$ 进行平凡着色 Ψ_0，即对每个顶点着上互不相同的颜色。然后构造一个部分着色序列 $\{\Psi_j\}$，$j \ge 0$。每产生一个部分着色 Ψ_j，FINDSET 就检查是否满足

$$Q(\Psi_j) \ge c_0 k / \log k$$

若是满足，则输出着某种颜色的顶点集合 C，C 已满足 $|C|+|N(C)|\geqslant c_0 k/\log k$；否则，它将对某些着色顶点进行去色，并将一些已着色的顶点集合并成为一个较大的顶点集，再对合并后的顶点集赋给新的颜色．这样就产生了一个新的部分着色 Ψ_{j+1}．显然 $|\Psi_j|\geqslant|\Psi_{j+1}|$，$j=0,1,\cdots$。这些工作将由后面将要介绍的函数 REDUCE 来完成．

为了达到去色的顶点数尽可能的少，且使合并后的顶点集尽可能大．我们采用了一种匹配技术．先将两个着色集合 C 和 C' 进行合并，使得 $|C\cap N(C')|$ 或 $|C'\cap N(C)|$ 尽可能的小；然后去掉 $C\cap N(C')$ 或 $C'\cap N(C)$ 的颜色．这可从 $B_{\Psi,h}$ 的图 $B^c_{\Psi,h}$ 找出一个最大匹配 M 来完成．也就是说，对 $B^c_{\Psi,h}$ 中的每一个匹配对 (C,C')，均要进行合并、去色工作．

对每一匹配的顶点对 $(C,C')\in M$ 来讲，其操作是：若 $|C\cap N(C')|<|C'\cap N(C)|$，则对集合 $C\cap N(C')$ 的顶点去色；否则对集合 $C'\cap N(C)$ 的顶点去色．然后对 $C\cap C'$ 中剩下的顶点着新的颜色，而对那些未匹配的顶点集仍着原来的颜色，这样就产生了 G 的一个新的部分着色．

要从 $B^c_{\Psi,h}$ 中寻找一个最大匹配 M，首先需构造辅图 $B_{\Psi,h}$，这可由函数 BUILD 完成；然后根据 $B_{\Psi,h}$ 很容易地构造出 $B^c_{\Psi,h}$；再由 $B^c_{\Psi,h}$ 找出一个最大匹配 M．后两项工作是由函数 MATCH 完成的．这样，FINDSET 可描述如下：

算法11.8 FINDING A VERTICES SET

输入：图 $G(V,E)$ 的边集合 E 和顶点子集 $A\subseteq V$；

输出：图 $G[A]$ 的一个独立集 C，$C\subseteq V$．

function FINDSET(A);
  begin
    (1) $k\leftarrow|A|$; $H\leftarrow G[A]$;
    (2) $\Psi\leftarrow$ trivial coloring of H;
    (3) **while** $Q(\Psi)<c_0 k/\log k$ **do**
      (3.1) $p\leftarrow|\Psi|$; $h\leftarrow c_1 k/(p\log k)$;
      (3.2) $B_{\Psi,h}\leftarrow$ BUILD(Ψ,h);
      (3.3) $M\leftarrow$ MATCH($B_{\Psi,h}$);
      (3.4) $\Psi\leftarrow$ REDUCE(Ψ,h,M)
      endwhile;
    (4) $C\leftarrow$ some vertex set such that $|C|+|N(C)|\geqslant c_0 k/\log k$;
    (5) FINDSET $\leftarrow C$
  end.

在进一步给出详细算法描述之前，我们给出一些定义．设 L 是一个已按关键字函数排好序的有序表，我们把 L 中具有相同关键字值的最大子表称为 L 的一个区

间 (Interval)。每个有序表可简单地看作是由许多区间连接而成的。

一个二元组 (r,s) 的字典序小于另一个二元组 (r',s') 当且仅当 $r<r'$ 或 $r=r'$ 且 $s<s'$。显然，无向图的边是一个二元组。

3. **函数BUILD**

函数 BUILD 用于构造辅图 $B_{\Psi,h}(V_1, E_1)$，它的输入是图 $G[A]$ 的一个部分着色 Ψ 和一个常数 $h>0$，其中 V_1 是颜色的集合，E_1 的计算涉及到 d_{ij} 的计算。为了计算 d_{ij}，BUILD 首先创建一个由 $G[A]$ 的边组成的表 L，然后对 L 按字典序进行排序，再把排序后的表中元素（每个元素是一条无向边）的每个端点的着色数作为关键字，对 L 再按字典序进行排序。之后在 L 上定义区间是端点着同一种颜色的最大子表。令 L_{ij} 是端点着第 i 种颜色和第 j 种颜色的区间。对每个区间 L_{ij} 来说，若以着第 j 种颜色的顶点作为关键字，对 L_{ij} 进行排序，则排序后 L_{ij} 所含的子区间① 数目即是 d_{ij} 的值。类似地，对 L_{ij} 按着第 i 种颜色的顶点进行排序，排序后的子区间数目即是 d_{ji} 的值。

为了确定一个排序表的各个区间，可以把表中每个元素的关键字，同它的左邻或右邻元素的关键字值进行比较，若相等，则给予一种特殊标识；否则，将该元素标识为一个新区间的开始或结束。然后用路径折叠技术对每个区间的成员进行标识（比如用关键字标识）。

BUILD的实现细节如下：

算法11.9 BUILDING A GRAPH

输入：图 $G[A]$ 的部分着色 Ψ 和正的常数 h；

输出：辅图 $B_{\Psi}(V_1,E_1)$。

function BUILD(Ψ,h)；

begin

(1) $E_1 \leftarrow \phi$；

(2) 对 $G[A]$ 的边按字典序进行并行排序，设排序后的有序表为 L；

(3) 对 L 以元素端点的着色数为关键字按字典序进行排序；

(4) $T(1) \leftarrow 1$；　/ * 标识数组，两个“1”之间的元素都属于同一区间 * /

(5) **for each** i: $1 \leqslant i \leqslant m_l$ **pardo**　/ * m_l 表示 $G[A]$ 边数 * /

　　if $L(i)$ 端点着色数 $\neq L(i-1)$ 端点着色数

　　　　then $T(i) \leftarrow 1$

　　　　else $T(i) \leftarrow 0$

　　endif

　endfor；

(6) **for each** interval L_{ij} **pardo**　/ * i,j 是区间内，边的两端点着色数 * /

①在这里，子区间定义为：以着第 j 种颜色的顶点为关键字排序后组成的子表。

(6.1) 以着第 j 种颜色顶点为关键字将 L_{ij} 排序，设排序后的表为 $L_{ij}^{(1)}$；

(6.2) $d_{ij} \leftarrow L_{ij}^{(1)}$ 中区间的个数；

(6.3) 以着第 i 种颜色顶点为关键字将 L_{ij} 排序后的表为 $L_{ij}^{(2)}$；

$d_{ji} \leftarrow L_{ij}^{(2)}$ 中区间的个数；

(6.4) **if** $(d_{ij} > h)$ **and** $(d_{ji} > h)$ **then** $E_1 \leftarrow E_1 \cup \{(i,j)\}$ **endif**

endfor

end.

因为整个 BUILD 最多涉及 $O(m)$ 个元素排序，由 Cole 算法可知，在 SIMD－EREW PRAM 上，需 $O(\log m) = O(\log n)$ 时间、$O(m)$ 处理器[7]。所以整个函数 BUILD 的执行需 $O(\log n)$ 时间、$O(m)$ 处理器。

4. 函数　MATCH

函数 MATCH 的输入是无向图 $B_{\Psi,h}$，输出是 $B_{\Psi,h}^c$ 的一个最大匹配 M。对算法的要求是：(1) 去色的顶点数应该尽可能的少；(2) 使尽可能多的顶点集进行合并。因为 $B_{\Psi,h}^c$ 中任意一条边 (C,C')，C 与 $N(C')$ 或 C' 与 $N(C)$ 的交集中元素个数至少有一个小于等于 h，所以删除的顶点数只能在 $B_{\Psi,h}^c$ 中有边关联的顶点之间进行．这样做就可满足要求 (1)。为了满足要求 (2)，我们需在 $B_{\Psi,h}^c$ 中寻找一个最大匹配 M．直观地讲，寻找一个图的最大匹配可归结为寻找它的线图的极大独立集．因而用直观方法是无法解决这个问题的．我们采用一个找 $B_{\Psi,h}^c$ 最大匹配的新方法．具体地讲，首先构造一个 $p = |\Psi|$ 个顶点的完全图 K_p，对 K_p 的边用 x 种颜色进行着色，使得同一顶点的关联边着不同的颜色，其中若 p 为奇数，则 $x = p$；若 p 为偶数，则 $x = p - 1$。结果是，图 K_p 的所有边分别被划分到边集合 P_0，P_1，…，P_{x-1} 某一个当中去，这种边的划分应使得每个集合 P_i 所含的边数相等 $(0 \leqslant i \leqslant x-1)$。将 K_p 的边 (i,j) 划分到集合 $P_{\mathrm{Index}(i,j)}$ 中去（Index 作为一个函数将在下面描述）．同样地，我们用 x 种颜色按相同的划分策略对图 $B_{\Psi,h}$ 的所有边进行着色，着色后不同颜色的边将划分到集合 PB_0，PB_1，…，PB_{x-1} 的某一个当中．显然 $PB_i \subseteq P_i (0 \leqslant i < x)$。令 PB_t 是 $\{PB_i |\ 0 \leqslant i < x\}$ 中边数最少的一个集合，则在图 $B_{\Psi,h}^c$ 中对应的一个最大的边匹配为 $M = P_t - PB_t$。下面给出 MATCH 的实现细节．

算法11.10　**FINDING A MAXIMAL CARDITICAL MATCHING**

输入：无向图 $B_{\Psi,h}$ 的边集合；

输出：$B_{\Psi,h}^c$ 的一个最大匹配 M。

```
function Index(i, j, p);
  begin
        if p is odd    then Index ← (i + j) mod p
                       else if j = p − 1   then Index ← 2i mod (p − 1)
                                           else Index ← (i + j) mod (p − 1)
                            endif
        endif
  end;

function MATCH(B_{Ψ,λ});      /* 主过程 */
   begin
   (1)  M ← φ;    p ← |Ψ|;
   (2)  if p is even   then x ← p − 1  else x ← p  endif;
   (3)  for each i: 0 ≤ i < x  pardo
           P_i ← φ;  PB_i ← φ
        endfor ;
   (4)  for each i,j : 1 ≤ i,j ≤ p pardo
        (4.1)  l ← Index(i, j, p);
        (4.2)  P_l ← P_l ∪ {(i,j)};
        (4.3)  if (i,j) ∈ E_1   then PB_l ← PB_l ∪ {(i,j)}  endif
                        /* (i,j)是B_{Ψ,λ}的边 */
        endfor ;
   (5)  for each l: 0 ≤ l < x  pardo
            find a set PB_l such that  |PB_l| = min{|PB_l| 0 ≤ l < x};
         endfor ;
   (6)  M ← P_l ∩ (P_l − PB_l);
   (7)  remove all but ⌈p(c_1 − c_0)/2c_1⌉ edges from M
  end.
```

引理 11.8　在 SIMD－EREW PRAM 上，计算 $B^c_{\Psi,\lambda}$ 的最大匹配，算法 11.10 需 $O(\log n)$ 时间、$O(m)$ 处理器。

证明　在算法 11.10 中，第 (1)～(2) 步仅需 $O(1)$ 时间及处理器；第 (3) 步至多需 $O(1)$ 时间、$O(n)$ 处理器；第 (4) 步是为第 (5)～(6) 步的计算而设计的。事实上没必要用 $O(p^2)$ 处理器，仅需 $O(m)$ 处理器执行第 (4.3) 步即可。也就是说，没有必要计算出所有的 P_l $(0 \leqslant l < x)$。因而这一步需 $O(\log n)$ 时间、$O(m)$ 处理器；第 (5) 步的实现如下：先将 $B_{\Psi,\lambda}$ 的边按关键字 Index 进行排序，设排序后的有序表为 L_1；接着是找出 L_1 的所有区间；

然后应用路径折叠技术计算每个区间所含的元素个数；最后选出所含元素个数最少的那个区间即为 PB_t。排序可采用 Cole 算法完成，需 $O(\log n)$ 时间、$O(m)$ 处理器。

设 $\text{Index}(i, j, p) = t$。不难看出：使用 $O(n)$ 处理器在常量时间内可找出所有满足约束的二元组 $\{(i, j) | \text{Index}(i, j, p) = t\}$，显然 $P_t = \{(i, j) | \text{Index}(i, j, p) = t\}$。又因 $M = P_t \cap (P_t - PB_t)$ 且 $PB_t \subseteq P_t$，故 $M = P_t - PB_t$。第 (6) 步可通过排序实现：首先将 P_t 与 PB_t 合并成一个表 L；然后应用 Cole 算法对 L 按字典序排序；再去掉重复的元素（二元组），剩下的有序表即是 M。因此，第 (6) 步至多需 $O(\log n)$ 时间、$O(m)$ 处理器；第 (7) 步可将有序表元素向表头移动，以填补前面删除的元素；然后只保留从表头开始的前面 $\lceil p(c_1 - c_0)/2c_1 \rceil$ 个元素，这一步可用前缀计算方法来实现[9]。即第 (7) 步至多需 $O(\log n)$ 时间、$O(m)$ 处理器。因此匹配函数 MATCH 的每次执行至多需 $O(\log n)$ 时间、$O(m)$ 处理器。

5. 函数 REDUCE

若 $(C,C') \in M$，即 C 和 C' 是一个匹配对，则在集 C 中着第 l 种颜色的每个顶点需要知道集 C' 内每个顶点的颜色。要在 SIMD – EREW PRAM 上完成这一任务，这里使用了一个过程 BROADCAST。它的工作是将着第 l_i 种颜色的每个顶点广播消息 m_i。

具体来讲，过程 BROADCAST 的输入是由二元组 (l_i, m_i) 组成的表 L_1，其中：l_i 表示着第 l_i 种颜色（即着色数），若 $i \neq j$，则 $l_i \neq l_j$；m_i 是要广播的消息。BROADCAST 先创建另一个由着色顶点形成的表 L_2，每个着色顶点 v 形成 L_2 的一个元素 (l_v,v)，l_v 是 v 的着色数；然后将 L_1 和 L_2 合并成表 L_3，并以着色数为关键字对 L_3 按字典序进行排序，当 L_1 和 L_2 中的元素具有相同的着色数时，约定 L_1 的元素排在 L_2 的元素前面；最后应用路径折叠技术，在排序后的 L_3 中每个区间上实现消息广播，其结果是：着第 l_i 种颜色的每个顶点将收到一个消息 m_i，$1 \leqslant i, j \leqslant |\Psi|$。

算法11.11 BROADCASTING IN A LINKED LIST

输入：顶点子集 $A \subseteq V$，以及 A 中元素 v 所着的颜色 $l(v)$；

输出：A 中元素的着色，即更新后的着色。

procedure BROADCAST(M, Ψ, A, L_1)；

begin

(1) create a list $L_2 = \{(l_v,v) \mid l_v \geqslant 1, v \in A\}$;

(2) form a list L_3 by concatenating L_1 and L_2;

(3) sort L_3 by the first coordinate of each element;

implement broadcasting at each interval by applying path doubling technique on L_3

end.

显然，算法 11.11 在 SIMD－EREW PRAM 上需 $O(\log n)$ 时间、$O(n+m)$ 处理器.

函数 REDUCE 除了决定对哪些顶点去色、以及对一些不同着色顶点集进行归并外，还需对着色的顶点的着色数重新给予连续的编号，且编号始于 1，以便下一次执行 MATCH 时使用.

函数REDUCE的实现细节如下:

算法11.12 REDUCING IN A GRAPH

输入：匹配M和交集元素矩阵$D=\{d_{ij}\}$;

输出：对每个$v\in V$进行颜色更新，更新后的颜色为$l(v)$.

```
function REDUCE(Ψ, h, M);
  begin
    (1) L1 ← φ;  L2 ← φ;
    (2) for each i, j : (i, j)∈M pardo
          if d_ij < d_ji
            then form a record(j ,m(j)),where m(j) ← i;
                 L1 ← L1 ∪ {(j, m(j))}
          endif
        endfor;
    (3) call BROADCAST (M, Ψ, A, L1);
    (4) for each v : (v∈A) and v receiving a color l(v) pardo
            if v is adjacent to a vertex of color l(v)
                then decolor v
                else change the color of v to l(v)
            endif
        endfor;
    (5) sort the vertices by their (new) color;
    (6) number the colors in use;
    (7) create a list L, its element(l, n(l)), where l is old color number, and n(l)
        is new color number;
    (8) call BROADCAST (M, Ψ, A, L);
    (9) each colored vertex changes its color to the message it received
    end .
```

引理 11.9 在 SIMD－EREW PRAM 上，算法 11.12 至多需 $O(\log n)$ 时间、$O(n+m)$ 处理器.

证明 算法 11.12 的第 (1) 步需 $O(1)$ 时间及处理器；第 (2) 步因 $B^{c}_{\Psi,h}$ 的顶点数为 $|\Psi|$

$\leqslant n$，而 M 是 $B^c_{\Psi,h}$ 的一个匹配，故 $|M| \leqslant \lfloor n/2 \rfloor$，所以第 (2) 步至多需 $O(\log n)$ 时间、$O(n)$ 处理器；第 (3) 步调用过程 BROADCAST 需 $O(\log n)$ 时间、$O(n+m)$ 处理器；第 (4) 步实现如下：先将收到颜色消息的顶点的邻接边形成一个表 L，然后对 L 按字典序排序，在每个区间上检查另一端点是否亦着了接收到的颜色消息，并应用路径折叠技术于每个区间，检查是否存在收到同一种颜色消息，若是的话，则此顶点去色；否则，此顶点改变它的颜色为接收的颜色。这些都涉及排序以及路径折叠技术，故最多需 $O(\log n)$ 时间、$O(n+m)$ 处理器；第 (5)～(7) 步类似第 (4) 步，它们至多需 $O(\log n)$ 时间、$O(n+m)$ 处理器；第 (8) 步同第 (3) 步类似，需 $O(\log n)$ 时间、$O(n+m)$ 处理器；第 (9) 步需 $O(1)$ 时间、$O(n)$ 处理器；所以整个算法 11.2 需 $O(\log n)$ 时间、$O(m+n)$ 处理器。

11.4.3 算法的复杂性分析

我们的目标是要证明：若使用 $O(n+m)$ 处理器，算法 11.8 能在 $O(\log^2 n)$ 时间内构造图 $G[A]$ 的一个独立集 C 且 $|C \cup N(C)| > c_0 k / \log k$，这里 c_0 是一个正常数，$k = |A|$。若能证明这一点，则不难看出，极大独立集算法调用算法 11.8 共 $O(\log^2 n)$ 次，故整个求极大独立集算法仅需 $O(\log^4 n)$ 时间、$O(n+m)$ 处理器。

首先我们分析算法 11.8 的计算复杂性。令 Ψ_i 是该算法中 while 循环体第 i 次执行前的 Ψ 值，令 $p_i = |\Psi_i|$，Δ_i 是 B_{Ψ_i,h_i} 的最大顶点数，$h_i = c_1 k / p_i \log k$，c_1 是一个正的常数。

设在 B_{Ψ_i,h_i} 中，最大度数顶点 v 所在的着色顶点集是 C，那么，一方面因 v 的度数为 Δ_i，所以 v 与其它 Δ_i 个着色顶点集中的顶点相关联，因而 $N(C)$ 至少含有 $\Delta_i h_i$ 个顶点；另一方面，由算法 11.8 可知，$|N(C)| < c_0 k / \log k$，所以 $\Delta_i h_i < c_0 k / \log k$，即 $\Delta_i c_1 k / p_i \log k < c_0 k / \log k$，故 $\Delta_i < c_0 p_i / c_1$。这就意味着 $B^c_{\Psi_i,h_i}$ 的每个顶点的度数至少为 $(c_1 - c_0)p_i / c_1$，故 $B^c_{\Psi_i,h_i}$ 的边数大于等于 $(c_1 - c_0)p_i^2 / 2c_1$，$c_1 > c_0$。又因为算法 11.10 将 $B^c_{\Psi_i,h_i}$ 的边集合划分为至多 p_i 个子集合，其中 $P_i - PB_i$ 是边最多的一个子集，由鸽巢原理知：$|P_i - PB_i| \geqslant (c_1 - c_0)p_i / c_1$。算法 11.10 将删除 $P_i - PB_i$ 中的一些边，最后返回一个匹配 M_i，且 $|M_i| = (c_1 - c_0)p_i / 2c_1$。当算法 11.12 创建 Ψ_{i+1} 时，去色的顶点数至多为

$$|M_i| c_1 k / p_i \log k = (c_1 - c_0)p_i / 2c_1 \cdot c_1 k / p_i \log k = (c_1 - c_0)k / 2\log k$$

且不同的着色顶点集个数减至 $a p_i$，其中 $a = (c_1 + c_0) / 2c_1$ 。初始化时部分着色 Ψ_0 有 $k = |\Psi_0|$ 种颜色，因此，算法 11.8 的 while 循环体至多执行 $-\log k / \log a$ 次。当不存在 Ψ_i，$|\Psi_i| > 1$ 且满足 $Q(\Psi_i) \geqslant c_o k / \log k$ 时，算法 11.8 去色的顶点数至多为

$$-(\log k / \log a)(c_1 - c_0)k / 2\log k = -(c_1 - c_0)k / 2\log a$$

结果，只剩下着一种颜色的顶点数至少为:

$$Q_0 = k - \left(-\frac{(c_1 - c_0)k\log k}{2\log a\log k} \right) = k - (c_1 - c_0)k / 2(-\log a)$$

若我们适当选择c_0及c_1，使得对某个确定的 $\varepsilon > 0$，有

$$-\log a = \log(\frac{2c_1}{c_1 + c_0}) > \frac{c_1 - c_0}{2c_1(1 - c_0)}(1 + \varepsilon)$$

那么

$$Q_0 > \varepsilon k / (1 + \varepsilon) > c_0 k / \log k$$

故对足够大的k，有$Q_0 > c_0 k / \log k$.

定理 11.9 SIMD－EREW PRAM 上，计算一个无向图 $G(V,E)$，$|V| = n$，$|E| = m$ 的极大独立集，算法 11.7 需 $O(\log^4 n)$ 时间、$O(n + m)$ 处理器.

证明 因算法 11.8 的 while 循环体需执行 $O(\log n)$ 次，而每次执行此循环体时，调用算法 11.10 和算法 11.12，分别需 $O(\log n)$ 时间、$O(n + m)$ 处理器. 求极大独立集的算法 11.7 需调用算法 11.8 $O(\log^2 n)$ 次，故整个极大独立集算法 11.7 需 $O(\log^4 n)$ 时间、$O(n + m)$ 处理器。

11.5 小 结

本章重点介绍了极大独立集问题的并行算法. 计算图的 MIS 的串行算法非常的简单，而它的并行算法设计则采用了完全不同的设计策略，且比串行算法复杂得多. 本章除了介绍 MIS 问题的并行算法外，还引入了随机并行算法的概念，介绍了怎样去构造一个随机算法的样本空间，以及叙述了将一个随机算法转换成确定算法的基本原则. 在本章重点介绍的两个并行算法中，第一个算法是由它的随机算法转换而成的，而后一个算法则完全是一个确定的算法. 应当注意，我们在这里不仅讨论了 MIS 并行算法设计问题，而且还间接地介绍第二章描述过的迭代改进设计技术，这种技术在并行算法设计中，具有普遍的指导意义.

有关极大独立集 (MIS) 问题的并行算法，除了上述介绍的以外，Alon 等人也曾独立地建议了一个随机并行算法[8]，在 SIMD－CRCW PRAM 上，他们算法的期望运行时间为 $O(\log n)$，使用的处理器数为 $O(m\Delta)$，Δ 是图中顶点的最大度数; Goldberg 等人利用 Cole 等人发明的硬币投掷技术[9] 建议了一个 $O(n^\alpha)$ 时间、$O(n)$ 处理器的 MIS 算法[10]，其中 α 是大于 1／2 的某个任意的数; Goldberg 等人在图顶点度数为常量的情况下，在 SIMD－EREW PRAM 上，曾建议了一个 $O(\log^* n)$ 时间、$O(n)$ 处理器的算法[11]；另外 Hen 考虑了平面图这一特殊情况的 MIS 问题，在 SIMD－CRCW PRAM 上建议了一个 $O(\log^2 n)$ 时间、$O(n)$ 处理器算法[12]，He 将 Luby 算法

应用到平面图上，其确定算法需 $O(\log n^2 n)$ 时间、$O(n^3)$ 处理器，因而 He 算法在平面图情况下改进了 Luby 算法使用的处理器数。

参 考 文 献

[1] Karp R M, Wigderson A. A Fast Parallel Algorithm for the Maximal Independent Set Problem, *Proc. 16th ACM Sympo. of Theory of Computmg*, 1984, 266–272

[2] Luby M. A Simple Parallel Algorithm for the Maximal Independent Set Probhem, *SIAM J. Comput.*, 15, 1986, 1036–1053

[3] Valiant L G. Parallel Computation, *in Proc. 7th IBM Sympo. on Math. Foundations of Computer Science*, 1982, 173–189

[4] Cook S A. Taxonomy of Problems with Fast Parallel Algorithms, *Inform. and Control*, 64,1985, 2–22

[5] Luby M. A Simple Parallel Algorithm for the Maximal Independent Set Problem, *Proc. 17th ACM Symp. on Theory of Computing*, 1985, 1–10

[6] Goldberg M, Spencer T. A New Parallel Algorithm for the Maximal Independdnt Set Problem, *SIAM J. Comput.*, 18(2), 1989, 419–427

[7] Cole R. Parallel Merge Sort, *Proc. 27th Annu. IEEE Sympo. on Poundations of Computer Science*, 1986, 511–516

[8] Alon N, Babai L, Itai A. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem, *J. Algorithms*, 7, 1986, 567–583.

[9] Cole R , Vishkin U. Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques fot Designing Parallel Algorithms, *Proc. 18th Annu. ACM Sympo. on Theory of Computing*, 1986, 206–219

[10] Goldberg M. Parallel Algorithms for These Graph Problems, *Congr. Numer.*, 54, 1986, 111–121

[11] Goldberg A V, Plotkin S A. Parallel (Δ+1)–Coloring of Constant–Degree Graphs, *Inform. Proc. Lett.*, 25, 1987, 241–245

[12] He X. A Nearly Optimal Parallel Algorithm for Constructing Maximal Independent Set in Planar Graphs, *Theoretical Computer Science*, 61, 1988, 33–47

[13] Rabin M O. *Probabilistic Algorithms*, in Algorithms and Complexity New Directions and Recent Results, Edited by, J. F. Traub 1976, 21–39

[14] 陈国良. 并行算法——排序和选择，合肥：中国科学技术大学出版社，1990

第十二章　图着色的并行算法

图着色问题是指对无向图顶点或边进行着色，使得相邻的顶点或与同一个顶点关联的边着不同颜色且整个图用尽可能少的颜色。这一问题具有广泛的应用背景，如任务调度，资源分配，VLSI 布线和测试等，都可归约成为图着色问题。因此，在图论应用中图着色问题一直受到广泛的重视。近年来，随着并行处理技术的进一步发展，人们研究出许多图的并行着色算法。本章我们将介绍其中一些有关图的顶点及边着色的并行算法。

12.1　图的顶点着色的并行算法

12.1.1　常数度图的着色算法

Goldberg 等人[1,2]基于 SIMD-EREW PRAM 模型，对顶点最大度数 Δ 是常数的一类图，建议了一个$(\Delta+1)$-着色的并行算法，他们的算法需 $O(\log^* n)$①时间、$O(n)$处理器。本节我们介绍这一算法。

1. 算法的基本思想

对一个图进行合法着色是要使得相邻顶点着不同的颜色。具体来讲，首先给图 G 顶点赋予初始着色，令 c 是 G 的初始着色，计算 c 的一种简单方法是：把与每个顶点相联的处理器编号作为该顶点的初始着色数，这样，每个顶点 v 都着了同其它顶点不同的颜色 $c(v)$；然后从初始着色出发，不断调整每个顶点的着色，使整个图的颜色数减少。

算法的思想是：对每个顶点建立一个差异表作为该顶点新的着色。差异表中含有它和邻接顶点之间的颜色差别。整个算法就是重复地构造各个顶点的差异表，不断地改进顶点的着色。差异表中每个元素是一个有序对，其中：有序对的第一分量是位置下标，它表示顶点的颜色与它的某个邻接顶点颜色第一次在此位置上不相同；有序对的第二分量是该位置的顶点差异表上用二进制数表示的编号值。下标可以用一个长为 $\lceil \log L \rceil$ 的二进制数表示，其中 L 是目前颜色数的十进制长度。由于图的顶点最大度数 Δ 是一个常数，因而新的着色数的二进制长度为 $O(\log L)$。

下面给出一个使用常数种颜色（也许较 $\Delta+1$ 大）对常数度图着色的并行算法。算法工作如下，首先给 G 赋予一个合法的初始着色，然后对图 G 循环地进行着色，每次循环都减少总的着色数目。对每个顶点 v 以及 v 的每个邻接顶点 w，算法找出 $c(v)$ 与 $c(w)$ 第一位不同的下标 i_w，同时构造一有序对 $<i_w,\ c(v,\ i_w)>$，其中 $c(v,\ i_w)$表示 $c(v)$的第 i_w 位的值。因为初始着色的合法性，所以这样的下标总是存在的。

2. 算法的形式化描述

---

① $\log^* x = \min\{i \mid \log^{i} x \leqslant 2\}$

算法 12.1　VERTEX–COLORING OF GRAPHS WITH CONSTANT DEGREE

输入：无向图的边集合$\{(u,v) \mid (u,v) \in E\}$；

输出：每个顶点 $v \in V$ 着一种颜色 $c(v)$ 且 $|c(V)|$ 为常量。

```
procedure Color__Constant__Degree__Graph(V);
  begin
  (1)  L ← ⌈logn⌉;
  (2)  for each v: v∈V  pardo
    (2.1)  c(v)←PE__ID(v);        /* PE__ID(v)是和v对应的处理器编号 */
    (2.2)  N(v)←{w | (v,w)∈E};
      endfor;
  (3)  while L > Δ ⌈logL+1⌉ do
    (3.1)  for each v: v∈V  pardo
      (3.1.1) for k←1 to |N(v)| do   /* N(v,k)表示N(v)的第k个元素 */
                w_k←N(v,k)
      (3.1.2)   i_k←min{i | c(v,i)≠c(w_k, i)};
      (3.1.3)   b_k←c(v,i_k)
              endfor;
      (3.1.4) for k←|N(v)| to Δ do
      (3.1.5)   i_k←0;
      (3.1.6)   b_k←c(v,0)
              endfor;
      (3.1.7) c(v)←i_1b_1i_2b_2…i_Δb_Δ   /* 计算新颜色 */
          endfor;
    (3.2)  L←Δ ⌈logL+1⌉
        endwhile
  end.
```

3. 算法的复杂性分析

定理 12.1　在 SIMD–EREW PRAM 上，用常数种颜色对一个无向图 G 进行着色，当图的最大度数 Δ 是常量时，算法 12.1 需 $O(\log^* n)$时间、$O(n)$处理器。

证明　由算法 12.1 可知，第(1)步需 $O(1)$时间及处理器；第(2.2)步因为 G 的最大度数 Δ 是常量，所以第(2)步需 $O(1)$时间、$O(n)$处理器；第(3.1.2)步涉及下标计算，应用 Cole 等人技术[5,6]，这一步可在 $O(1)$时间内完成，其它各步显然至多需 $O(1)$时间、$O(n)$处理器。

现在我们证明第(3)步的 **while** 循环体至多执行 $O(\log^* n)$次。令 $L=\lceil \log n \rceil$，即处理器编号需要的二进制位数。令 L_k 表示第 k 次循环后颜色代码的位数。我们用归纳法来证明：对 $k=1$ 时，有

$$L_1 = \Delta \lceil \log L + 1 \rceil \leqslant 2\Delta \lceil \log L \rceil$$

设 $\lceil \log L \rceil \geqslant \lceil \log \Delta \rceil + 2$。假定对某个 $k-1$ 时，有

$$L_{k-1} \leqslant 2\Delta \lceil \log^{(k-1)} L \rceil$$

$$\lceil \log^{k} L \rceil \geqslant \lceil \log\Delta \rceil + 2$$

那么,

$$L_k = \Delta \lceil \log^{k-1} L + 1 \rceil \leqslant \Delta(\lceil \log(2\Delta \log^{(k-1)} L \rceil + 1) \leqslant 2\Delta \lceil \log^{k} L \rceil$$

因此，只要 $\lceil \log^{k} L \rceil \geqslant \lceil \log\Delta \rceil + 2$，就有 $L_k \leqslant 2\Delta \lceil \log^{k} L \rceil$。由此可见，颜色数的位数 L_k 减少，直到 $O(\log^{*} n)$ 次循环后降到常数 $2\Delta \lceil \log\Delta + 2 \rceil$ 或更小时，算法终止。因此整个算法需 $O(\log^{*} n)$ 时间、$O(n)$ 处理器。

定理 12.2 算法 12.1 能正确地给图 $G(V,E)$着色。

证明 我们用归纳法证明算法 12.1 能正确地给 G 着色。初始化时，每个处理器有一个唯一的编号，因此，初始着色是合法的，下面我们证明若在一次循环前着色是合法的，则执行循环后计算出的新的着色也是合法的。若在一次循环之前着色是合法的，那么每对邻接顶点 v 及 w，至少存在一个下标 i_w，$c(v)$与 $c(w)$在第 i_w 位不同。对每一个顶点 v，算法为它构造一个新的颜色可以看作由形式为 $<i_w, c(v, i_w)>$ 的有序对组成的表，每个邻接顶点形成一有序对。为了使这些差异表组成一个新的合法着色，任何两个邻接顶点 v 及 w 必须至少有一个不同的有序对。假定由 v 构造的差异表是:

$$(<i_1, c(v, i_1)>, <i_2, c(v, i_2)>, <i_3, c(v, i_3)>, \cdots, <i_\Delta, c(v, i_\Delta)>)$$

由顶点 w 构造的差异表是:

$$(<j_1, c(w, j_1)>, <j_2, c(w, j_2)>, \cdots, <j_\Delta, c(w, j_\Delta)>)$$

若对某个 k，$i_k \neq j_k$，则 $c(v)$ 与 $c(w)$ 至少有一位不同。反之，设对所有 $1 \leqslant k \leqslant \Delta$，$i_k = j_k$，令$<i_k, c(v, i_k)>$是由 v 构造的差异表中对应边(v, w)的有序对，根据 i_k 的定义，$c(v, i_k) \neq c(w, j_k)$，又 $i_k = j_k$，因此，$c(v, i_k) \neq c(w_1, j_k)$。故每对邻接顶点，构造出的差异表总是不同的。

应该指出的是，上述建议的算法不仅解决了求解无向图的着色问题，实际上，它引入了一种新技术一**破对称技术**(Symmetry Breaking Technique)，这种技术蕴含着许多其它的用途。

12.1.2 常数度图着色算法的应用

前面介绍的着色算法可应用到求无向图的极大独立集问题，这里再描述一个求极大独立集(MIS)的并行算法，它将调用着色算法 12.1。

算法 12.2 MIS OF GRAPH WIIH CONSTANT DEGREE

```
    输入: 无向图 G(V,E);
    输出: 极大独立集 I, I⊆V.
  procedure MIS__Constant__Degree__Graph (V, I);
    begin
  (1)  I←φ;    A←V;
  (2)  call__Color__Constant__Degree__Graph(V);    / * 调用着色算法 12.1 * /
  (3)  while A≠φ do
    (3.1)  I'←{v | v∈A and c(v) is same};
```

/ * 从 A 中选出着同一种颜色的顶点形成的集合 * /

(3.2) $A \leftarrow A-(I' \cup N(I'))$;

(3.3) $I \leftarrow I \cup I'$

endwhile

end.

定理 12.3 在 SIMD-EREW PRAM 上，求一个最大度数 Δ 为常量的无向图的极大独立集，算法 12.2 需 $O(\log^* n)$时间、$O(n)$处理器

证明 由算法 12.2 可知，第(1)步需 $O(1)$时间及处理器；第(2)步由定理 12.1 可知，需 $O(\log^* n)$时间、$O(n)$处理器；整个第(3)步的 **while** 循环体执行次数完全由图的着色数决定。因为图 G 仅用了常数种颜色着色，所以 **while** 循环执行 $O(1)$次。在 **while** 循环体内，第(3.1)～(3.3)步需 $O(1)$时间、$O(n)$处理器。因此计算无向图 G 的 MIS 需 $O(\log^* n)$时间、$O(n)$处理器。

现在我们利用算法 12.2，给出一个度数最大为 Δ 的无向图的顶点着 Δ+1 种颜色的并行算法，Δ+1 着色算法的形式化描述如下：

算法 12.3 COLOR-Δ+1-GRAPH WITH Δ-DEGREE

输入：最大度数 Δ 是常量的图边集合$\{(u,v)|\ u,v\in V\}$；

输出：每个顶点 $v\in V$ 着一种颜色 $c(v)$且$|c(v)|\leqslant \Delta+1$.

procedure Color__(Δ+1)__Graph (V,E);

begin

(1) $A \leftarrow V$;

(2) **while** $A\neq\phi$ **do**

(2.1) **call** MIS__Constant__Degree__Graph(A,I);

(2.2) Color each $v\in I$ with a new color;

(2.3) $A \leftarrow A-I$

endwhile

end.

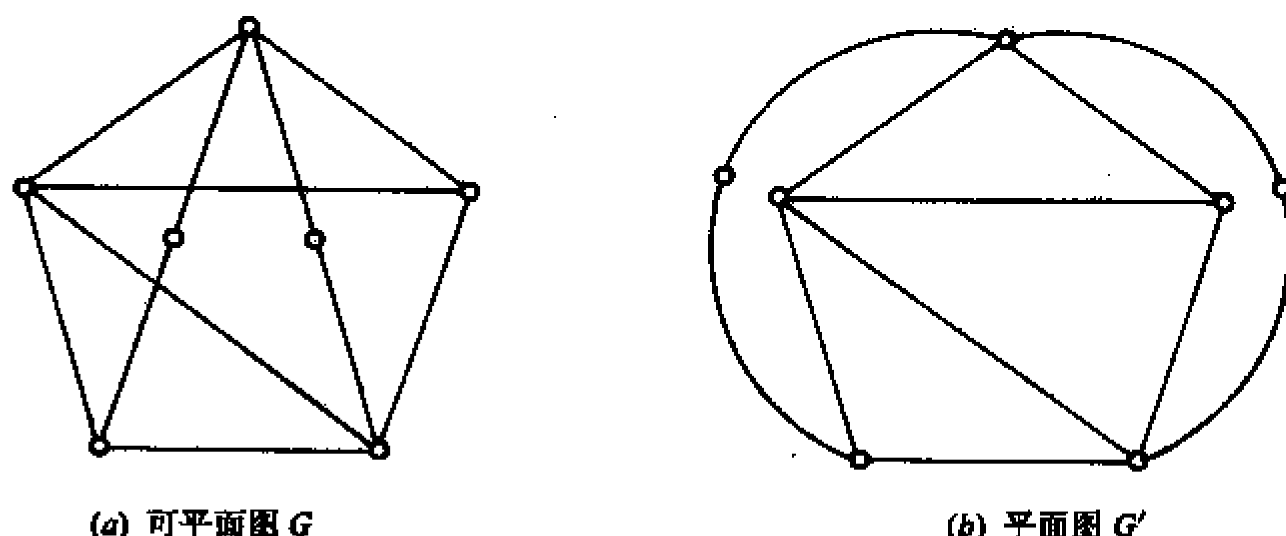
上面的算法显然能对 G 进行(Δ+1)着色。因为在算法 12.3 中每次循环即找出输入图的极大独立集 I，并对 I 内顶点用新的颜色着色，然后把 I 从原图 G 中删除。若一个顶点 v 在某次循环后没有被删除掉，那么它的邻接顶点至少有一个在这次循环中被删除了。因此，在每次执行循环后，由剩余顶点导出的子图的顶点最大度数至少减少 1，所以，至多 Δ+1 次循环后算法终止，结果给出图 G 的一个(Δ+1)着色。

定理 12.4 在 SIMD—EREW PRAM 上，对一个最大度数 Δ 是常量的图 $G(V,E)$, $|V|=n$ 进行(Δ+1)着色的并行算法 12.3，需 $O(\log^* n)$时间、$O(n)$处理器。

证明 由于算法 12.3 的 **while** 循环体至多执行(Δ+1)次，而每次调用极大独立集算法需 $O(\log^* n)$时间、$O(n)$处理器；第(2.2)～(2.3)步需 $O(1)$时间、$O(n)$处理器。因此整个算法需 $O(\log^* n)$时间、$O(n)$处理器。

12.1.3 平面图5–着色并行算法

一个图 G，若能把它画在平面上，且除端点外任意两条边均不相交，则称 G 是可以**嵌入**(Embedding)平面的。如果一个图 G 是可以嵌入平面的，那么 G 为**可平面图**(Planar Graph)。可平面图在平面上的一个嵌入称为一个**平面图**(Planar Graph)。如图 12.1 所示，(*a*)所示的图 G 是一个可平面图，(*b*)所示的图 G' 是 G 的一个平面嵌入，即平面图。



(*a*) 可平面图 G　　(*b*) 平面图 G'

图 12.1 可平面图和平面图示例

设 G 是一个平面图，图的边所包围的一个区域，其内部既不含图的顶点也不含图的边，这样的区域称为 G 的一个**面**(Face)。面的边界就是包围该面的诸边所构成的回路。

设 $G(V,E)$ 是一个连通平面图，且 $|V|=n, |E|=m$，则有 $m\leqslant 3n-6$。

设 $G(V,E)$ 是一个连通的平面图，令 $c(v)$ 是顶点 $v\in V$ 的颜色，对 $W\subseteq V$，令 $c(W)$ 是 W 内着色顶点所着颜色的集合。由颜色 α 及颜色 β 的顶点导出的子图记为 $G_{\alpha\beta}$。令 $N(v)$ 是 v 的邻接顶点集，$N(v)=\{u|(v,u)\in E\}$。令 $d(v)$ 表示顶点 v 的度数。

首先我们介绍 Naor 给出的一个简单算法[3]，在 SIMD–EREW PRAM 上，他建议了一个 $O(\log^5 n)$ 时间、$O(n^3)$ 处理器的并行算法。

1. 算法的基本原理

Naor 算法主要利用了平面图一些特殊性质，下面我们介绍这些性质。

引理 12.1 设 $G(V,E)$ 是一个无向连通平面图，且 $|V|=n$，则 G 中度数小于 7 的顶点数大于 $n/6$。

证明 若 G 中度数小于 7 的顶点数 $\leqslant n/6$，则 G 的度数之和至少为 $n/6+7\times 5n/6=6n$，而平面图的度数之和至多为 $6n-12$，这导致矛盾，故引理成立。

引理 12.2 设 $G(V,E)$ 是一无向连通平面图，U 是度数小于 7 的顶点集合，则由 U 导出的子图 $G[U]$ 的每个极大独立集 I，其规模至少为 $|U|/7$。

证明 对 U 中任一顶点 $v\in U$，若 $v\notin I$，则 v 是极大独立集 I 中某个顶点的邻接顶点，因此 $|I|\geqslant |U|/7$。

上面两个引理导出了平面图的一个 7–着色算法。由引理 12.1 可知，若每次从平面图中删除度数小于 7 的顶点，则删除的顶点个数是整个图顶点数的常数倍①。因此，经过 $O(\log n)$ 次删除后，原来的图将成为至多含一个顶点的图。引理 12.2 告诉我们，由度数小

①常数因子大于0小于1。

于 7 的顶点导出的子图，其极大独立集大小是整个图顶点数的常数倍。因此经过 $O(\log n)$ 次计算剩下的图的极大独立集，可以变为至多含一个顶点的独立集。而极大独立集内的顶点是可以着同一种颜色的。

2. 平面图的 7–着色并行算法

算法 12.4 VERTEX COLORING OF PLANAR GRAPHS WITH 7–COLORS

输入：图 G 的邻接矩阵 A;

输出：每个顶点 $v \in V$ 着的颜色 $c(v)$，且 $|c(V)| \leq 7$.

```
procedure  Seven_Coloring (V);
  begin
  (1)  V'←V;
  (2)  while  V'≠φ  do
  (3)     U←{v | d(v)<7, v∈V'};
  (4)     find a maximal independent set I of graph G[U];
  (5)     call  Seven_Coloring (V'−U∪N(U));
  (6)     set c(v), v∈I, d(v)≤6, so that c(v)≠c(w), (v,w)∈E;
       endwhile
  end.
```

定理 12.5 在 SIMD–EREW PRAM 上，对一平面图 $G(V,E)$，$|V|=n$ 进行 7–着色算法 12.4，需 $O(\log^3 n)$时间、$O(n^3)$处理器。

证明 由引理 12.1 可知，过程 Seven_Coloring 需递归 $O(\log n)$次。又由定理 11.8 知，每次递归时执行第(4)步至多需 $O(\log^2 n)$时间、$O(n^3)$处理器；而第(6)步仅需 $O(1)$时间、$O(n)$处理器。因此，整个算法需 $O(\log^3 n)$时间、$O(n^3)$处理器。

在给出平面图的 5–着色算法之前，我们先证明一个重要定理。

定理 12.6 任意给定的一个平面图 $G(V,E)$，$|V|=n$，$|E|=m$，若对 V 着色，则最多需要 5 种颜色。

证明 对顶点数 n 用归纳法证明。

(a) 当 $n \leq 5$ 时，定理显然成立。

(b) 假定$n=k(k>5)$时定理成立。现考察$n=k+1$。因为$|V|>5$，所以至少存在一个顶点$v \in V$，且 $d(v) \leq 5$。如果不是这样，由于$|E|=m$，我们有$\sum_{v \in V} d(v)=2m$，又由$|V|=n$和$d(v) \geq 6$，得$\sum_{v \in V} d(v) \geq 6n$，故$m \geq 3n > 3n-6$，与平面图性质矛盾。因此在 G 中一定存在一个顶点 v 且 $d(v) \leq 5$。从 G 中删除 v，则根据归纳假定，对图 $G[V-\{v\}]$定理成立。再把 v 加到 $G[V-\{v\}]$中，若 $d(v)<5$，则与 v 的邻接顶点数$\leq$ 4，因而有一种剩余的颜色可用于 v 着色，得出一个 5–色图 G。

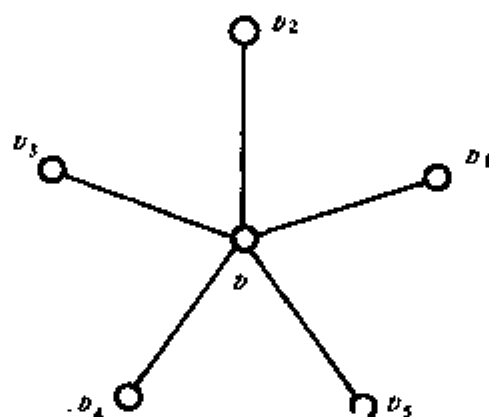


图 12.2 $|v|=6, d(v)=5$ 的着色图

若 $d(v)=5$，设与 v 邻接的顶点按逆时针排列为 v_1，v_2，v_3，v_4，v_5，它们分别着不同颜色 c_1，c_2，c_3，c_4，c_5，如图 12.2 所示。令 H 是 $G[V-\{v\}]$中所有着 c_1 与 c_3 色的顶点集合。图中的 $H=\{v_1, v_3\}$，F 为 $G[V-\{v\}]$中所有着 c_2 及 c_4 色的顶点集合。图中的 $F=\{v_2, v_4\}$。

若 v_1 与 v_3 属于顶点集合 H 所导出子图的两个不同连通分支中，将 v_1 和 v_3 所在连通分支中的颜色对调，并不影响 $G[V-\{v\}]$的正常着色。具体地讲，设 v_1 所在的连通分支中，顶点集合为$\{v_1^i\}$，v_3 所在集合为 $\{v_3^i\}$。若 $|\{v_1^i\}|\leqslant|\{v_3^i\}|$，则将 $\{v_1^i\}$ 的颜色置换成 $\{v_3^i\}$的颜色，这样，v_1 的着色为 c_3。然后在 v 上着色 c_1，否则在 v 上着色 c_3。故 G 是 5−色的。

如果 v_1 与 v_3 属于顶点集合 H 所导出子图的同一个连通分支中，那么从 v_1 到 v_3 必有一条路径 P，P 上的每个顶点都是着 c_1 色或 c_3 色。该路径与边(v,v_1)和边(v,v_3)一起构成一条回路 L，L 包围了 v_2 或 v_4，但不能同时包围 v_2 和 v_4，故 v_2 与 v_4 分别属于顶点集 F 所导出子图的两个不同连通分支中。同理在包含 v_2 和 v_4 的连通分支中将颜色对调并不影响 $G[V-\{v\}]$的正常着色，比如将 v_2 换成着色 c_4，这样顶点 v_2 与 v_4 都着 c_4 色，故对 v 着 c_2 色，即可对 G 进行 5−色着色。

现在讨论如何将 $G(V,E)$的颜色数从 7 降至 5 以及如何将 5−着色从 $G-I$ 扩展到图 G。若顶点 v 的邻集 $N(v)$只有少于 5 种颜色的着色，则剩下的一种颜色可以对 v 着色。否则，因为 $G-I$ 是 5−着色的，故 $N(v)$正好着了 5 种颜色。又因 G 的颜色数是 7，所以 v 最多只有 6 个邻接顶点。于是 v 的邻接顶点中至少有 4 个顶点 w_1，w_2，w_3，w_4 已着色且w_i $(1\leqslant i\leqslant 4)$与 v 的任何其它邻接顶点着不同颜色。由定理 12.5 可知，在这四个顶点中，存在一对着 α，β 颜色的顶点，它们属于 $G_{\alpha\beta}$ 的两个不同的连通分支中。我们通过 v 首先确定出 w_1,w_2,w_3，w_4，然后确定这四个顶点中哪一对着 α，β 颜色，令其分别属于 $G_{\alpha\beta}$ 的不同连通分支中（若这样的顶点对不止一对，则任取一对）。由于仅有 10 种颜色对，故必有一对顶点着色 α，β 且其邻接顶点着这两种颜色的未着色顶点数目至少为$|I|/10$。设邻接顶点着 α，β 颜色的未着色顶点集为 J 且$|J|=k$。每一个着 α 或 β 色的顶点将由它所在 $G_{\alpha\beta}$ 的连通分支标识。若在与 $v\in J$ 邻接的两个顶点在 $G_{\alpha\beta}$ 的两个连通分支中，仅将其中一个的 α，β 颜色对调，结果给 v 留下一种空闲的第 5 种颜色。

为确定在哪个连通分支中可以将 α，β 颜色同时对调，我们建立一种多重图 H。H 中的顶点 C_i 是 $G_{\alpha\beta}$ 中的一个连通分支。若 J 中有一个顶点 v 同时与 C_i 及 C_j相连接，则顶点 C_i，C_j之间有边相连（J 中每个顶点恰好与 $G_{\alpha\beta}$ 的两个连通分支邻接），因此，H 中有 k 条边。显然，多重图 H 是一个平面图，我们可以用 7 种颜色对之并行着色。着同一种颜色的各连通分支是独立的，可以对这些连通分支内顶点的着色同时进行对调。H 中顶点度数之和为 $2k$，因此至少存在着一种颜色的顶点度数之和为 $2k/7$。由此可见，J 中能自行着色的顶点数目是 I 中顶点数目小于 1 的常数倍。

3. 算法的形式化描述

算法 12.5 VERTEX COLORING OF PLANAR GRAPHS WITH 5−COLORS

输入：无向连通平面图 $G(V,E)$的邻接矩阵；

输出：对每个顶点 $v\in V$ 着上一种颜色 $c(v)$，且相邻顶点着不同颜色，颜色数目至多是 5.

```
procedure Five_Coloring (V);
begin
```

(1) **while** $V\neq\phi$ **do**

(2) $U\leftarrow\{v \mid d(v)<7, v\in V\}$;

(3) find a maximal independent set I of graph $G[U]$;

(4) **call** Five_Coloring ($V-(I\cup N(I))$);

(5) /＊将5−着色扩充到删除的顶点集＊/

(5.1) 对每个 $v\in I$，若 $|c(N(v))|<5$，则至少有一种空闲颜色对 v 着色，否则做：

(5.2) **while** ($v\in I$) **and** $c(v)=0$ **do** /＊$c(v)=0$ 表示 v 没着色＊/

(5.2.1) I 中每个未着色顶点 v 确定一对不属于 $G_{\gamma\delta}$ 的同一连通分支的顶点（分别着 γ 及 δ 颜色）；

(5.2.2) 令 α，β 为这样一个颜色对，它至少被 I 中 $|I|/10$ 未着色顶点选用，用 J 表示选用 α，β 的未着色顶点集，且令 $|J|=k$；

(5.2.3) 建立多重图 H 如下：顶点 C_i 是 $G_{\alpha\beta}$ 的一个连通分支，若 J 中存在一个顶点 $v\in J$，v 与 C_i 及 C_j 相连，则 C_i 与 C_j 之间有边相连；

(5.2.4) **call**_Seven_Coloring($\{C_i \mid C_i$ 是 $G_{\alpha\beta}$ 的一个连通分支$\}$)；

(5.2.5) 在 H 中选出度数和至少为 $2k/7$ 的着同一颜色顶点集 g；

(5.2.6) 在 $G_{\alpha\beta}$ 中，将属于 g 的连通分支中的 α，β 颜色对调；

(5.2.7) 结果每个 $v\in J$，存在一空闲颜色对之着色

endwhile

endwhile

end.

定理 12.7 在 SIMD−EREW PRAM 上，对一个无向平面图 5 着色的算法 12.5，需 $O(\log^5 n)$时间、$O(n^3)$处理器.

证明 由算法 12.5 可知：第(4)步的递归调用需执行 $O(\log n)$次。而第(5.1)步仅需 $O(1)$时间、$O(n)$处理器；第(5.2.1)步应用并行求连通分支的算法 4.3，需 $O(\log^2 n)$时间、$O(n^2)$处理器；第(5.2.2)步可用 Cole 排序算法实现，每个未着色顶点 v 形成一个或多个三元组(v,α,β)，其中 α，β 是它的邻接顶点所着的颜色，然后以(α,β)为关键字排序，计算排序后最长的区间内的顶点即为 J 的成员，这一步需 $O(\log n)$时间、$O(n)$处理器；第(5.2.3)步仅需 $O(1)$时间、$O(n)$处理器；由定理 12.5 知，第(5.2.4)步需 $O(\log^3 n)$时间、$O(n^3)$处理器；第(5.2.4)～(5.2.7)步需 $O(1)$时间、$O(n)$处理器；而(5.2.1)～(5.2.7)至多执行 $O(\log n)$次；由定理 11.8 可知，第(3)步需 $O(\log^2 n)$时间、$O(n^3)$处理器。因而整个算法需 $O(\log^5 n)$时间、$O(n^3)$处理器.

推论 12.1 在 SIMD−EREW PRAM 上，对一个无向平面图 5−着色需 $O(\log^7 n)$时间、$O(n^2)$处理器。

证明 对算法 12.4 及算法 12.5 中求极大独立集算法换用 Goldberg 算法，由定理 11.9 知，计算一个 n 个顶点无向图的极大独立集需 $O(\log^4 n)$时间、$O(n+m)$处理器，因而 5−着色算法需 $O(\log^7 n)$时间、$O(n+m)$处理器.

推论 12.2 在 SIMD−EREW PRAM 上，对一个无向平面图进行 5−着色，需 $O(\log^5 n)$时间、$O(n^2)$处理器。

证明 因所着色的图是平面图且辅图 H 也是平面图，使用 He 的平面图极大独立集

算法（见本书第 11 章参考文献[12]），在 SIMD−CRCW PRAM 上，求 n 个顶点平面图的极大独立集需 $O(\log^2 n)$时间、$O(n)$处理器，它可在 $O(\log^3 n)$时间内在 $O(n^2)$处理器的 SIMD−EREW PRAM 上模拟实现，因而平面图 5−着色需 $O(\log^5 n)$时间、$O(n^2)$处理器.

12.1.4 平面图 5−着色最优的并行算法

Hagerup 等人[4]基于 SIMD−EREW PRAM 模型，建议了一个 $O(\log n\log^* n)$时间、$O(n/\log n\cdot\log^* n)$处理器的平面图 5−着色最优的并行算法．本节我们将介绍这个最优算法．

1. 一些基本概念

设 $G(V,E)$是一个简单无向图．对 $u,v\in V$ 且$(u,v)\notin E$，u 和 v 的粘合（Identification）是一种操作，此操作用一个新顶点 z 替换掉 u 及 v 和它们在图中关联的边，z 恰与 $V-\{u,v\}$ 中那些在原图中同 u 和 v 关联的顶点关联．多于两对不邻接顶点的粘合可类似地定义，或看成两个顶点粘合的多次应用．当 C 是一个独立集时，我们用$\langle C\rangle$表示 C 中所有顶点的粘合．

所谓图 G 的一个平面图嵌人，是一个函数 Ψ，这函数将 G 的顶点映射到 $\mathbf{R}^2$ 内的不同点上，且 G 的每条边 $e=(u,v)\in E$ 映射成 $\mathbf{R}^2$ 内的一条若当(Jordan)曲线，该若当曲线是从 $\Psi(u)$到 $\Psi(v)$的．对任意一条边 $e=(u,v)\in E$，它满足：

$$\Psi(e)\cap\Psi(V)\cup\Psi((E-\{e\}))\subseteq\{\Psi(u),\Psi(v)\}$$

即若 G 有平面嵌人，则 G 映射到平面的曲线不相交．

设 G 的一个平面嵌人是 Ψ，对顶点 $w\in V$，其邻集为 $N(w)=\{u_1,\cdots,u_k\}$，则围绕 $\Psi(w)$的若当曲线 $\Psi((w,u_1))$，$\cdots$，$\Psi((w,u_k))$按特定的环形次序(Cyclic Order)出现，即以逆时针方向围绕 $\Psi(w)$扫描时它们所出现的次序。可简单地说，在平面嵌人 Ψ 内 w 的邻接顶点按环形次序 $u_1,\cdots,u_k$ 围绕 w 出现。Ψ 的一个若当曲面是 $\mathbf{R}^2-\Psi(V\cup E)$的一个连通区域．

下面讨论无向平面图 $G(V,E)$的 5−着色最优并行算法．

2. 算法的基本原理

Hagerup 等人建议的 5−着色最优并行算法，其基本思想是：算法始于一个平面图 $G_0=G$，随后的各个连续阶段，算法产生一个平面图序列 G_0，G_1，G_2，$\cdots$，G_i，$\cdots$，每个 G_i 的大小至多是它的直接前趋$(i\geqslant 1)$的大小的 c 倍，$0<c<1$ 且是常数．因此，经过 $O(\log n)$个阶段后，对剩下图的着色是一件非常简单的事。

对 $i\geqslant 0$，G_{i+1} 是 G_i 的顶点经过许多次归约(Reduction)得来的．所谓顶点 w 的归约，是指删除 w 和粘合 w 的一些邻接顶点。

设 w 是 G 的一个顶点．若我们粘合 w 的足够多的邻接顶点，使 w 的度数降低到至多为 4，令得到的图为 G'，则将图 $G'-\{w\}$的 5−着色扩展至图 G 的 5−着色是一件容易的事．首先恢复 w 的邻接顶点的粘合，令在这一过程中创建的每一新顶点维持它的颜色。然后将 w 粘回到原来的位置，根据上述着色约定，w 的邻接顶点至多着 4 色，因此第 5 种颜色可用于对 w 着色．

为了使得 $G'-\{w\}$是可 5−着色的．我们必须保证它仍是一个平面图．进而证明对每个度数至多为 6 的顶点执行归约，以保持 $G'-\{w\}$的平面性．但是象这样的归约是不易确定的。因此，我们必须粘合足够大的顶点集，使得这个顶点集可在常量时间内找出来。

所有从 G_i 产生 G_{i+1} 的归约是并行执行的。归约顶点集 W 的选择必须倍加小心。首

先，它必须使得 G_{i+1} 仍然是平面图；其次，W 内的顶点和平面图 G_i 的顶点，其表示法必须保持某种独立性；最后，W 的大小应是整个顶点集大小的常数倍．这样，我们选择的归约顶点集 W 才是所有顶点的归约，且可在常量时间内完成．

给出一个图 G，在 G 中执行归约，我们形式地定义为集合 $r=\{w,C_1,\cdots,C_s\}$，其中 w 是 G 的一个顶点，C_1，…，C_s 是 $N(w)$中两两不相交的独立子集。顶点 w 称为集合 r 的中心(Center)．所谓执行归约 r 是的指删除 r 的中心 w 且对得到的图使用粘合操作 $<C_1>,\cdots,<C_s>$．

对一个固定的常量 $K\geqslant 12$，若某个顶点至多有 K 个邻接顶点，则它是小的(Small)，否则称它是大的(Large)。

所谓图 G 的一个顶点 w 是可归约的(Reducible)，若它满足下列条件之一:

(1) $d(w)\leqslant 4$（即顶点 w 的度数 $\leqslant 4$）;

(2) $d(w)=5$ 且 w 至多有一个大的邻接顶点;

(3) $d(w)=6$且w的所有邻接顶点都是小的．而$N(w)$导出的子图是一个哈密顿图，即 $N(w)$生成的子图是一条简单回路。

若下列条件之一成立:

(1) $d(w)\leqslant 4$ 且 $r=\{w\}$;

(2) $d(w)=5$，对 w 的某一对不同的小邻接顶点$\{x,y\}$，有 $r=\{w,\{x,y\}\}$;

(3) $d(w)=6$，或者

(a) $r=\{w,C\}$，存在 $C\subseteq N(w)$且$|C|=3$，或者，

(b) $r=\{w,\{x_1,y_1\},\{x_2,y_2\}\}$，对某个不同的顶点对，$x_i\in N(\omega)$，$y_i\in N(\omega)$，$i=1$，2．在 $N(w)$导出子图的哈密顿回路上的环形次序为 x_1，y_1，x_2，y_2。

则称以可归约顶点 w 为中心的归约 r 是安全的(Safe)。

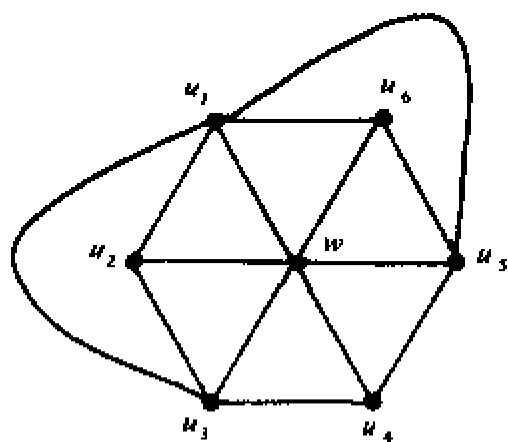


图 12.3　若 u_1 与 u_3 和 u_5 邻接，则$\{u_2,u_4,u_6\}$是一个独立集

平面图的 5-着色并行算法是并行执行以可归约顶点为中心的安全归约．算法主要基于下述六个引理．引理 12.3 证明了安全归约的存在性；引理 12.4 保证了安全归约保持图的平面性；引理 12.5 及引理 12.6 论证了可归约顶点数是大的；引理 12.7 证明了在顶点度数为常量时，求 n 个顶点图的一个大的独立集，可以在 $O(\log^* n)$时间内实现；由前面诸条引理，引理 12.8 最终陈述了以可归约顶点为中心的、具有常数倍个可归约顶点的归约，可在 $O(\log^* n)$时间内实现．

引理 12.3　设顶点 w 是平面图 G 的一个可归约顶点，且已知 w 的邻接表和它的小邻接顶点，那么存在一个以 w 为中心的安全归约，可用一个处理器在 $O(1)$时间内完成归约．

证明　$d(w)\leqslant 4$，引理 12.3 显然成立．若 $d(w)=5$，则可以标识 w 的两个邻接顶点，且它们是不相邻的．显然这两个顶点一定存在．不然的话，G 将含一个 K_5 子图，与 G 是平面图矛盾．若 $d(w)=6$，令 w 的六个邻接顶点分别为 u_1，u_2，…，u_6，且令它们在 $N(w)$导出的哈密顿回路上的环形次序为 u_1，u_2，…，u_6．

若 $N(w)$含有 3 个顶点的独立集，则可以直接标识它们．否则我们假定 G 含有边

(u_1, u_3)，那么 G 不可能含有边(u_1, u_5)，不然的话，$\{u_2, u_4, u_6\}$将形成一个独立集（如图 12.3 所示）. 因此$\{w, \{u_1, u_5\}, \{u_2, u_4\}\}$是以 w 为中心的一个安全归约.

引理 12.4 对平面图 G 执行安全归约后得出的图 G'仍是一个平面图。

证明 考虑安全归约的一种形式$\{w, \{x_1, y_1\}, \{x_2, y_2\}\}$. 在图 G 的任一平面嵌入内，不仅 x_1 与 y_1 而且 x_2 与 y_2 是相邻的，且在围绕 w 的环形次序上它们出现的次序为 x_1, y_1, x_2, y_2. G'的平面性见图 12.4, 其它情况的安全归约不难验证.

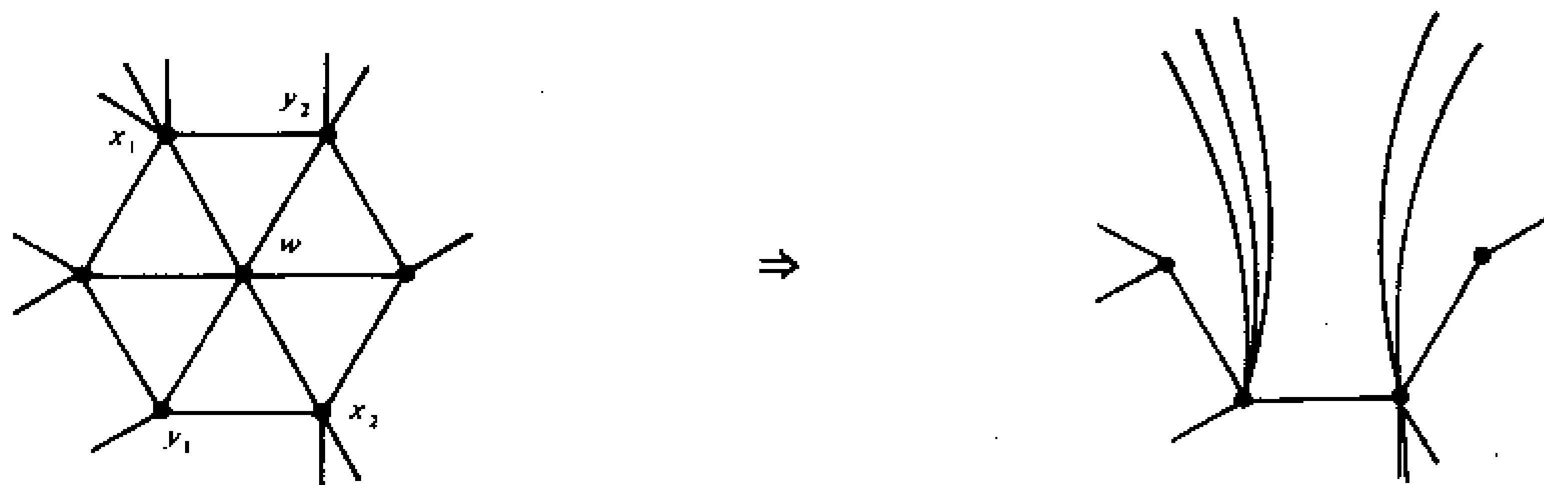


图 12.4 执行安全归约$\{w, \{x_1, y_1\}, \{x_2, y_2\}\}$

引理 12.5 设 $G(V,E)$, $|V|=n$, $|E|=m$ 是一个平面图，且 $Z=\{w \mid d(w)\geqslant 3, w\in V$,且 $N(w)$导出的子图不是哈密顿图$\}$，那么 $m\leqslant 3n-Z/4$。

证明 考虑 G 的任一平面嵌入，我们观察到 Z 的每个顶点限制在这个面上，它至少含四个不同的顶点. 现在对每个这样的面 F，给 G 加入一个新顶点 u_F，且加入连接 u_F 与 F 上所有顶点的边（若一个顶点在边界上不止一次出现，则 u_F 仅与其中一个有边相连），那么得到的图仍是一个平面图. 令 n', m' 分别表示添加的顶点数及边数，显然 $m'\geqslant 4n'$ 且 $m'\geqslant |Z|$。再对得到的平面图应用欧拉公式，则得 $(m+m')\leqslant 3(n+n')$，即

$$m\leqslant 3n+3n'-m'\leqslant 3n-m'/4\leqslant 3n-|Z|/4$$

引理 12.6 设 A 是在 N 个顶点的平面图 G 上可归约的顶点集，那么 $|A|\geqslant n/196$。

证明 令 x 是 G 的顶点最大度数。对 $j=0,1,2,\cdots,x$. 令 n_j 表示图 G 中度数为 j 的顶点数. 令$\hat{n}_5$ 表示度数是 5 的可归约的顶点数，$\bar{n}_6$ 表示度数是 6 的可归约的顶点数，$\hat{n}_6$ 表示度数是 6 且它的邻接顶点都是小的顶点数目. 显然 $\hat{n}_6\geqslant\bar{n}_6$. 故 $|A|=\sum_{j=0}^{4}n_j+\hat{n}_5+\bar{n}_6$.

令 $\varepsilon=6-2m/n$, $\varepsilon\geqslant 0$, 故 $2m=(6-\varepsilon)n$，或 $\sum_{j=0}^{x}j\cdot n_j=2m=(6-\varepsilon)\sum_{j=0}^{x}n_j$. 将含 n_5 的项分离出来，得到

$$(1-\varepsilon)n_5=\sum_{j=0}^{4}(j-6+\varepsilon)n_j+\sum_{j=6}^{x}(j-6+\varepsilon)n_j$$

所以，

$$n_5\geqslant\sum_{j=6}^{x}(j-6+\varepsilon)n_j-6\sum_{j=0}^{4}n_j \tag{1}$$

令 m' 是图 G 中有一个端点是大的边数。由$\hat{n}_5$ 及$\hat{n}_6$ 定义，$m' \geqslant 2(n_5 - \hat{n}_5) + (n_6 - \hat{n}_6)$。另一方面，$m' \geqslant \sum_{j=K+1}^{x} j \cdot n_j$。因此，$2(n_5 - \hat{n}_5) + (n_6 - \hat{n}_6) \leqslant \sum_{j=K+1}^{x} j \cdot n_j$，或者，

$$2\hat{n}_5 + \hat{n}_6 \geqslant 2n_5 + n_6 - \sum_{i=K+1}^{x} j \cdot n_j \tag{2}$$

将不等式(1)乘以 $\alpha = 15/8$ 加到不等式(2)上，结果为：

$$2\hat{n}_5 + \hat{n}_6 \geqslant (2-\alpha)n_5 + n_6 - \sum_{j=K+1}^{x} j \cdot n_j + \alpha \sum_{j=6}^{x} (j - 6 + \varepsilon) n_j - 6\alpha \sum_{j=0}^{4} n_j$$

$$= n_5 / 8 + (1 + \alpha\varepsilon) n_6 + \alpha \sum_{j=7}^{K} n_j + \sum_{j=K+1}^{x} ((\alpha - 1)(K+1) - 6\alpha) n_j - 6\alpha \sum_{j=0}^{4} n_j$$

由于$(\alpha - 1)(K+1) - 6\alpha \geqslant 1/8$，因此得

$$6\alpha \sum_{j=0}^{4} n_j + 2\hat{n}_5 + \hat{n}_6 \geqslant \frac{n_5}{8} + (1 + \alpha\varepsilon) n_6 + \frac{1}{8} \sum_{j=7}^{x} n_j \tag{3}$$

下面考虑两种情况：

情况 I：$\varepsilon \geqslant 1/32$。由于$n_6 \geqslant \hat{n}_6$，所以

$$6\alpha \sum_{j=6}^{4} n_j + 2\hat{n}_5 \geqslant \frac{n_5}{8} + \frac{\alpha n_6}{32} + \frac{1}{8} \sum_{j=1}^{x} n_j$$

上式两边乘以$32/\alpha \geqslant 8$，并加上$\sum_{j=0}^{4} n_j$，得：

$$193 \cdot |A| \geqslant (6 \times 32 + 1) \sum_{j=0}^{4} n_j + \frac{64}{2} \hat{n}_5 \geqslant \sum_{j=0}^{4} n_j + n_5 + n_6 + \sum_{j=7}^{x} n_j = n$$

引理12.6对这种情况成立。

情况 II：$\varepsilon < 1/32$。由于 $m = 3n - \varepsilon n / 2$。根据引理 12.5 得：$\hat{n}_6 - \bar{n}_6 \leqslant 2\varepsilon n$，因此由 (3) 得：

$$6\alpha \sum_{j=0}^{4} n_j + 2\hat{n}_5 + \bar{n}_6 \geqslant \frac{n_5}{8} + n_6 + \frac{1}{8} \sum_{j=7}^{x} n_j - 2\varepsilon n$$

且

$$(6\alpha + 1) \cdot |A| \geqslant (6\alpha + 1) \sum_{j=0}^{4} n_j + 2\hat{n}_5 + \bar{n}_6 \geqslant \left(\frac{1}{8} - 2\varepsilon\right) n \geqslant \frac{n}{16}$$

故 $|A| \geqslant n / 16(6\alpha + 1) = n / 196$。

为了下面讨论方便，现给出图 G 的数据结构。对顶点集 $V = \{1,2,\cdots,n\}$，令每个顶点 $u \in V$ 有一个**双向邻接链表**，链表中的每一结点恰好对应 u 的一个邻接顶点 v，除了含有 v 的标识外，还含有一个指针，它指向 v 邻接链表中对应 u 的结点。这种指针通常称为**交叉链** (Cross Link)。

引理 12.7 设 $G(V,E)$ 是一个 n 个顶点的无向图。且 G 的顶点最大度数 Δ 是常量，顶点的编号 $\leqslant q$。那么 G 至少含有 $n / 6^{\Delta}$ 个顶点的独立集，这个独立集可在 $O(\log^* q)$ 时间内使用 $O(n)$ 处理器计算出来。

证明 我们把 G 视作为一个有向图来考虑。即把 G 的每条无向边(u,v)看作为两条有向边 $<u,v>$ 和 $<v,u>$。对每个顶点 $u\in V$，指派一个处理器，且给 l 条出边，标号为 $1,2,\cdots,l$。令 E_j 是标号为 j 的边集合，为了消除集合 E_j 内的边引起的冲突，可执行下列程序段:

$V_0 \leftarrow V$;

for $j \leftarrow 1$ **to** Δ **do**

$V_j \leftarrow$ an independent set in the graph $G(V_{j-1}, E_j \cap (V_{j-1} \times V_{j-1})$ with $|V_j| \geqslant |V_{j-1}|/6$

endfor;

得到的 V_Δ 将是 G 的一个至少含 $n/6^\Delta$ 个顶点的独立集。剩下要证明：在具有 t 个顶点且最大出度为 1 的 G 的子图 G_1 中，可以在 $O(\log^* q)$时间内使用 $O(n)$处理器，找出一个至少含 $t/6$ 个顶点的独立集。

考虑这样一个子图 G_1，若删除 G_1 中入度$\geqslant 2$ 的顶点，则得到的图 H 是由一些没有公共顶点的简单路径和简单回路组成的。而且 H 至少含 $t/2$ 个顶点。我们考虑 H 中的任一简单回路和简单路径，因为顶点的最大度数为 2，它是一个常量，应用定理 12.3 知，寻找 H 的极大独立集需 $O(\log^* q)$时间、$O(n)$处理器；而 H 的极大独立集所含顶点数至少为$(1/3)(t/2)=t/6$。因此引理得证。

引理 12.8 设 A 是 n 个顶点的图 $G(V,E)$可归约的顶点集，G 的顶点编号限制在 $0\sim q$ 范围内。对每个 $u\in A$，令 r_u 是以 u 为中心的一个安全归约。那么对某个常数 $\gamma>0$，安全归约集合$\{r_u \mid u\in A\}$中至少有 $\gamma|A|$ 个归约可在 $O(\log^* q)$时间内使用 $O(n)$处理器同时执行。

证明 首先考虑执行一个安全归约。顶点 x_1，…，x_l 的粘合实现如下：先选择一个代表比如 x_1；再对 x_2，…，x_l 的邻接表中的结点重新命名，使其成为 x_1 邻接表中的结点，这只要将 x_1，…，x_l 的邻接表重新链接，形成 x_1 的新的邻接表即可；然后删除 x_2，…，x_l 和在 x_1 邻接表中重复出现的边，由于涉及粘合操作的顶点数目和它们关联的边均为常量，这一操作可用一个处理器在常量时间内完成。上述的安全归约可推广应用到删除一个顶点 w 及其邻接顶点（至多 6 个）。

为解决同时归约可能引起的归约不正确性，这里我们引入一个辅图 H，H 是一无向图，其顶点集为 A，它的每条边$(u，v)$是指归约 r_u 及 r_v 不能同时执行。对所有的 $u,v\in A$ 且 $u\neq v$，若有下列条件之一成立:

(1) 在 G 中存在一条从 u 到 v 的路径，其路径长度至多为 4，且在这条路径上不含大顶点；

(2) 存在小顶点 x 与 y，$x\in\{u\}\cup N(u)$，$y\in\{v\}\cup N(v)$，且 x 与 y 有一个公共的邻接顶点 w，在 w 的邻接表中，对应 x 与 y 的结点是紧挨在一起的。

显然 H 的最大度数也是常量。不难看到：因为归约顶点的邻接顶点数目至多为 6，且都为小顶点，故条件(2)可在 $O(1)$时间使用 $O(1)$处理器完成判定；至于条件(1)，因为路径长度至多为 4，所以也可用 $O(1)$处理器在 $O(1)$时间完成。因而构造辅图 H 需 $O(1)$时间、$O(n)$处理器。根据引理 12.8，存在一个常数 $\gamma>0$，使得 H 中大小至少为 $\gamma|A|$的一个独立集 W 可在 $O(\log^* q)$时间内计算出来。然后对每个 $w\in W$ 赋给一个处理器，对 r_w 同时执行归约。我们应该看到，这同串行地执行归约得到的图是一致的。且在以顶点 w 为中心的归约是在不同的存贮单元集（即无读写冲突）上进行的。为说明这一点，如果两个处理器 PE_1 和 PE_2 同时对一个邻接表 L 进行操作，那么，由条件(1)，L 必然是一个大顶点的邻接表，PE_1 和

PE_2 在 L 上的操作限制为修改或删除表的某些结点．在 L 中 PE_1 和 PE_2 不能同时删除或修改同一结点。因此，修改不会引起冲突．由条件(2)，L 甚至不含这样两个紧挨的结点，PE_1 删除其中的一个，PE_2 删除另一个．因此涉及到重置前趋及后继指针的删除也不引起冲突．

3. 算法的形式化描述

算法 12.6 THE OPTIMAL ALGORITHM OF PLANAR GRAPH'S FIVE–COLORING

输入：无向平面图 $G(V,E)$，用双向邻接表表示；

输出：每个项点 $v \in V$ 着一种颜色 $c(v)$，且$|c(V)| \leq 5$.

begin

(1) $G_0(V_0, E_0) \leftarrow G(V, E)$;

(2) $k \leftarrow \beta \lceil \log n \rceil$;

(3) **for** $i \leftarrow 0$ **to** $k-1$ **do**

  (3.1) **for** each u: $u \in V_i$ **pardo**

    Compute the set A_i of reducible vertices in G

   endfor;

  (3.2) **for** each u: $u \in A_i$ **pardo**

    Compute a safe reduction r_u centred at u

   endfor;

  (3.3) $G_{i+1}(V_{i+1}, E_{i+1})$ be a graph obtained from G_i by parallel executing at least $\gamma|A_i|$ of the reductions in $\{r_u \mid u \in A_i\}$

  endfor;

(4) Color graph $G_k(V_k, E_k)$;

(5) **for** $i \leftarrow k-1$ **downto** 0 **do**

  (5.1) Reconstruct G_i from G_{i+1};

  (5.2) Extend the coloring of G_{i+1} to a 5–coloring of G_i

  endfor

end.

其中 β, γ 均为大于 0 的常数，在常量因子问题上，它们的选取对算法效率的优劣影响很大．

定理 12.8 在 SIMD–EREW PRAM 上，对一个平面图 $G(V,E)$，$|V|=n$，$|E|=m$ 进行 5–着色，算法 12.6 需 $O(\log n \log^* n)$时间、$O(n)$处理器．

证明 我们通过分析算法 12.6 的每一步的计算复杂性来给出它的复杂性．算法的输入是平面图 G 的邻接表，为了使得 G 以双向链表表示且有交叉链存在，可以使用 Tarjan 的方法对边进行排序来实现．这样，形成 G 在算法中的数据结构需 $O(\log n)$时间、$O(m)=O(n)$处理器；算法的第(1)步需 $O(1)$时间、$O(n)$处理器；第(2)步仅需 $O(1)$时间及处理器；由引理 12.3 知，第(3.1)、(3.2)步需 $O(1)$时间、$O(n)$处理器；由引理 12.8 知，第(3.3)步需 $O(\log^* n)=O(\log^* n)$时间、$O(n)$处理器．故整个第(3)步需 $O(\log n \log^* n)$时间、$O(n)$处理器；由于 $|V_{i+1}| \leq |V_i| - \gamma|A_i| \leq (1-\gamma/196)|V_i|$，对 $i=0,1,\cdots$，适当地选择常

量 β. 保证 $|V_k| \leqslant 1$，从而使得 $G_k(V_k,\ E_k)$ 的着色非常简单。因此第 (4) 步至多需 $O(1)$ 时间、$O(n)$处理器；第(5)步类似第(3)步，故同第(3)步有相同的计算复杂性。从 G_{i+1} 着色扩展到 G_i 着色可在 $O(1)$时间内使用 $O(n)$处理器完成。因此，整个算法的时间复杂性为 $O(\log n \log^* n)$ 处理器复杂性为 $O(n)$。注意，当一个顶点 z 被分裂成两个顶点 x 和 y 时，x 和 y 继承顶点 z 的颜色。当一个顶点 w 被重新加入到图中时，因为 w 的邻接顶点至多用 4 种颜色着色，因此至少还有一种颜色可供 w 着色。显然从 G_{i+1} 着色扩展到 G_i 着色是一件非常容易的事，为了避免引起读冲突，每条边的端点必须带上它的颜色。

定理 12.9 在 SIMD-EREW PRAM 上，对一个无向平面图 $G(V,E)$进行 5-着色，可在 $O(\log n \log^* n)$时间内使用 $O(n/\log\log^* n)$处理器完成。

证明 构造 G 的双向邻接链表，在使用 $O(n/\log n \log^* n)$处理器时需 $O(\log n \log^* n)$时间。因为对 G 的边进行排序不能在 $O(\log n \log^* n)$时间完成，所以采用另一种简单方法构造交叉链。我们在每个邻接表上执行表计数算法[5]，一旦遇到顶点 v 的邻接表中的对应邻接顶点 u 时，将指向它的指针值置入 $T_{u,v}$ 单元，这样，计算所有交叉链需 $O(\log n \log^* n)$时间、$O(n/\log n \log^* n)$处理器；再对算法 12.6 的处理器谨慎地分配，比如用 Cole 等人的元素近似均匀分配算法[5,6]实现，则这种分配需 $O(\log n)$时间。且每一阶段使用的处理器数是按几何级数下降的。结果，我们得到一个无向平面图 5-着色的最优算法，此算法需 $O(\log n \log^* n)$时间、$O(n/\log n \log^* n)$处理器。

12.2 图的边着色的并行算法

在上一节中我们介绍了图的顶点着色算法，这一节我们将介绍边着色的并行算法。

一个图的边着色是对所有边给予着色，使得共用一个顶点的多条边着不同的颜色，而且希望使用尽可能少的颜色。给定一个无向图 $G(V,E)$，对它的边进行着色的最少颜色数目叫做 Chromatic 数，记作 $\chi'(G)$。令 G 的顶点最大度数为 Δ，显然 $\chi'(G) \geqslant \Delta$。然而，Vizing[7]证明了：若 G 是一个简单图，则 $\chi'(G) \leqslant \Delta+1$。若 G 是一个多重图(Multigraph)，Shannon[8]已证明了 $\chi'(G) \leqslant 3\lfloor \Delta/2 \rfloor$。

12.2.1 树的边着色并行算法

Gibbons 等人[9]基于 SIMD-CREW PRAM 建议了一个对树边着色的快速并行算法。这里将介绍他们的算法。

设 Δ 是一棵树 T 中顶点的最大度数。我们用 Δ 种颜色 $Q=\{0,\ 1,\ 2,\ \cdots,\ \Delta-1\}$对 T 进行着色。令 $x \in Q$，$y \in Q$，定义环和 $x \otimes y=(x+y) \bmod \Delta$，则有下面的引理。

引理 12.9 对任一元素 $x \in Q$，则 Q 中的 Δ 个元素 x，$x\otimes 1$，$x\otimes 2$，…，$x\otimes(\Delta-1)$均不相同。

证明 假定存在元素 $x\otimes i = x\otimes j$，且 $i<j$，$i,\ j \in\{0,\ 1,\ \cdots,\ \Delta-1\}$，则根据操作"$\otimes$"定义 $x\otimes i = r\cdot\Delta+k$，$x\otimes j = s\cdot\Delta+k$，因为 $i<j$，所以 $r\neq s$，故寻出 $x\otimes i \neq x\otimes j$，因此，$x$，$x\otimes 1$，$x\otimes 2$，…，$x\otimes(\Delta-1)$均不相同。

1. **算法的基本原理**

有关树并行计算的一个通用技术，就是在第八章介绍的 Tarjan 等人发展的欧拉遍历技术。利用这一技术，我们将树 T 以一个顶点为根的有向树形式存贮，对每个顶点，我们知道它的父亲和儿子。对树根附加一条入边且对这条边着色为 0，其它每个顶点 v 仅有一条进入边 $<F(v),v>$ （$F(v)$是 v 的父亲）和至多 $\Delta-1$ 条射出边。我们对 v 的所有射出边用$\{1, \cdots, \Delta-1\}$中的不同元素分别给予着色，设 $cs(v)$是从根开始（包括根的入边）到达 v 的路径上所有边着色的$\otimes$和。对每个顶点 v，我们对它的进入边 $<F(v),v>$着色 $cs(v)$，即 $c((F(v),v))\leftarrow cs(v)$。由引理 12.9 可知，这种对树 T 边的着色方法是一个合法的边着色方法。

2. **算法的形式化描述**

算法 12.7 EDGE COLORING OF TREES

输入：树 $T(V,E)$的邻接表；

输出：树 T 的每条边 $e\in E$ 的着色 $c(e)$，其中 $0\leqslant c(e)\leqslant \Delta$。

procedure Edge–Coloring__of__Trees(V,E);

begin

(1)(1.1) 调用算法 8.5，首先将 T 变成一个有向欧拉图；

(1.2) 对每个顶点的射出边着属于$\{1, 2, \cdots, \Delta\}$中互不相同的颜色，设着色数组为 label;

(1.3) 遍历有向欧拉图，令遍历后的表为 $L=(e_1, \cdots, e_{2m})$，T 的每条无向边 $(F(v), v)$在 L 中出现两次，即$<F(v), v>$ 和$<v, F(v)>$；

(2) 对 L 中的另一些边同时进行着色，即：label(v, $F(v)$)$\leftarrow$−label($F(v)$, v);

/ * 边$<F(v),v>$已在(1.2)步进行了着色。任何回路上的 label 的“$\otimes$”为 0 * /

(3) **for** each v: $v\in V$ **pardo**

cs(v)$\leftarrow$label(e_1)$\otimes$label(e_2) $\otimes\cdots\otimes$label(e_k)

endfor;

(4) **for** each v: $v\in V$ **pardo**

$c((F(v), v))\leftarrow$cs(v)

endfor

end.

定理 12.10 在 SIMD–CREW PRAM 上，对一棵树 $T(V,E)$，$|V|=n$，$|E|=m$ 的边着色，算法 12.7 需 $O(\log n)$时间和 $O(n)$处理器。

证明 根据算法 12.7 可知，第(1)步由定理 8.7 需 $O(\log m)=O(\log n)$时间和 $O(n+m)=O(n)$处理器($m=O(n)$)；第(2)步需 $O(\log n)$时间和 $O(n)$处理器；第(3)步需 $O(\log n)$时间和 $O(n)$处理器，这一步实际上是计算前缀的部分；第(4)步需 $O(1)$时间、$O(n)$处理器。故整个算法需 $O(\log n)$时间、$O(n)$处理器。

12.2.2 d–路图和 d–路二分图的基本概念

Lev 等人[10]基于 SIMD–EREW PRAM 建议了二分图边着色的并行算法。他们把置

换网络(Permutation Network)中的选路问题归约为图论上的图和二分图的边着色问题．下面我们将介绍他们的算法．

一个 n 个顶点 d–路图(n–vertex d–way Graph)是由顶点集$\{1,2,\cdots,n\}\times\{1,2,\cdots,d\}$和分枝集(Branch Set)组成，其中每个分枝是不同顶点的一个无序对，每个顶点至多出现在一个分枝中．

已知一个 d–路图，人们可以获得一个顶点度数至多为 d 的图．具体做法是：将每个顶点子集$\{(i,1),(i,2),\cdots,(i,d)\}$替换成一个顶点 i，将每个分枝$((i,j),(i',j'))$用边(i,i')替换，这样我们就得到一个一般的无向图，$1\leqslant i,\ i'\leqslant n$，$1\leqslant j,\ j'\leqslant d$．

一个 $2m$ 个顶点 d–路二分图($2m$–vertex d–way Bipartite Graph)是由顶点集$\{1,2,\cdots,m\}\times\{1,2,\cdots,d\}\times\{1,2\}$和分枝集所组成．每个分枝是不同顶点的一个无序对，这个无序对的一个顶点来自集合$\{1,2,\cdots,m\}\times\{1,2,\cdots,d\}\times\{1\}$，另一顶点来自集合$\{1,2,\cdots,m\}\times\{1,2,\cdots,d\}\times\{2\}$．每个顶点至多在一个分枝中出现．注意本节的 m 不是指图的边数．

我们使用 d–路图或 d–路二分图，是因为它们较对应的图或二分图含有更多的信息．显而易见，一个图或二分图很容易转换成对应的 d–路图或 d 路二分图．本节仅讨论 d–路图或 d–路二分图的边着色．一个 d–路图或 d–路二分图的着色是一个函数，这个函数给它的每个顶点赋给一种颜色 $j\in\{1,2,\cdots,d\}$．d–路图或 d–路二分图的一个顶点对应原来的图或二分图的 d 个顶点，它们将赋给不同的颜色．对 d–路图或 d–路二分图的一种着色是边着色，若每个分枝的两个端点都着上相同的颜色，则这种颜色就是对应图的边着色．

12.2.3 2–路图的边着色并行算法

首先我们讨论一个最简单的 2–路图的边着色算法．一个 2–路图是由许多不同的连通分支组成，每个连通分支或是一条无回路的路径或是一条回路．2–路图有 α–色的边着色当且仅当它不含奇数条边的回路（简称奇数边回路）．

1. 算法的基本原理

本节的算法或是给 2–路图的边着色或是检测奇数边回路．n 个顶点的 2–路图将用两个二维数组 I, J 表示．用 $I(i,j)\leftarrow i'$，$J(i,j)\leftarrow j'$，$I(i',j')\leftarrow i$ 和 $J(i',j')\leftarrow j$表示一条分枝$((i,j),(i',j'))$的存贮．若顶点(i,j)不在分枝中出现，则置 $I(i,j)\leftarrow 0$，$J(i,j)\leftarrow 0$，图的着色将用二维数组 Q 存贮，顶点(i,j)的颜色为 $Q(i,j)$．

若 2–路图存在一个边着色，则着色不唯一．这是因为对任一连通分支内的颜色可以进行对换．为使着色唯一，算法将给出**规范的边着色**(Canonical Edge Coloring)．顶点(i,j)的规范着色是指：若 i 是对应图的连通分支内标号最小的顶点，则 j就是 2–路（d–路）图的顶点(i,j)的颜色．

算法的基本策略是：对每个连通分支内的最小标号顶点的规范着色以指数的速率迅速传播到整个连通分支．在一个回路中，这种传播可以返回；若回路中含偶数条边，则不会引起什么后果；若回路中含奇数条边，则将引起写冲突．这样就可以检测奇数边回路存在．

2. 算法的形式化描述

算法 12.8　EDGE-COLORING 2-WAY GRAPHS

输入：2-路图的存贮结构：二维数组 I, J;

输出：顶点着色的二维数组 Q；若不存在边着色，则检测出奇数边回路。

```
procedure Edge_Coloring_2_Way_Graph(I, J, Q);
  begin
  (1) for each i: 1≤i≤n pardo        / * 初始化 * /
          for j←1 to 2 do
              I*(i, j)←I(i, j);
              J*(i, j)←J(i, j);
              P(i,j)←i;   / * P 存贮连通分支的最小顶点标号 * /
              Q(i,j)←j
        endfor
     endfor;
  (2) for s←1 to ⌈logn⌉ do
        for each i: 1≤i≤n pardo
          for j←1 to 2 do    / * 距顶点(i,j)距离为 2^s 的顶点(i*,j*) * /
            i*←I*(i,j);   j*←J*(i,j);
            if i*>0    / * 表示此顶点在一分枝中 * /
               then if P(i,j)>P(i*,j*)   / * (i*,j*)的当前连通分支标识较小 * /
                     then P(i,j)←P(i*,j*);
                          if s=1
                            then Q(i,j)←Q(i*,j*)   / * 初始着色 * /
                            else Q(i,j)←3-Q(i*,j*)
               / * (i, j)与(i*, j*)距离为偶，故(i, j)应与(i*, j*)着不同颜色 * /
                          endif
                     endif
            endif;
               / * 找出与(i,j)距离为 2^(s+1) 的顶点，若存在的话 * /
            I*(i, j)←I*(i*, 3-j*);
            J*(i,j)←J*(i*, 3-j*)
          endfor;
          for j←1  to 2  do
           / * 对每个分枝的顶点赋予当前连通分支的最小标号 * /
           if P(i,j)>P(i,+3-j)
                then      P(i, j)←P(i, 3-j);
                          Q(i, j)←3-Q(i, 3-j)
           endif;
                 / * 检测奇数边回路 * /
```

```
          if P(i, 1)=P(i, 2) and Q(i, 1)=Q(i, 2)
              /*对应图的顶点i两条边着相同颜色*/
              then  return('There exists odd cycle')
          endif
        endfor
      endfor
    endfor
  end.
```

算法 12.8 在执行 s 次循环后($1\leqslant s\leqslant\lceil\log n\rceil$), (1) 若存在一个顶点$(i^*,j^*)$, 它与$(i,j)$之间距离是 2^s, 则 $I^s(i,j)=i^*$, $J^s(i,j)=j^*$; 若没有这样的顶点, 则$I^s(i,j)=0$ 且 $J^s(i,j)=0$。(2) $P(i,j)$是与 i 距离至多为 2^{s-1} 的所有顶点的最小标号。(3) 若 i 是到它至多为 2^{s-1} 的所有顶点的最小标号, 则 $Q(i,j)$是顶点(i,j)的规范着色。

定理 12.11 在 SIMD−EREW PRAM 上, 对一个 n 个顶点的 2−路图进行边着色, 算法 12.8 需 $O(\log n)$时间、$O(n)$处理器。

证明 由算法 12.8 可知, 第(1)步需 $O(1)$时间、$O(n)$处理器; 第(2)步需 $O(\log n)$时间、$O(n)$处理器. 故整个算法需 $O(\log n)$时间和 $O(n)$处理器。

12.2.4 二次幂 d−路二分图边着色的并行算法

本节我们介绍 d−路二分图边着色算法, 应当指出, 这里仅考虑 d 是 2 的幂的情形。

一个 $2m$ 个顶点 d−路二分图将用两个三维数组 I 和 J 表示。用 $I(i,j,1)=i'$, $J(i,j,1)=j'$, $I(i',j',2)\leftarrow i$, $J(i',j',2)=j$来表示一个分枝$((i,j,1),(i,j',2))$。若顶点(i,j,k)不在任何一条分枝中出现, 则置 $I(i,j,k)\leftarrow 0$, $J(i,j,k)\leftarrow 0$。这样一个二分图的着色将存放在三维数组 Q 中, 顶点(i,j,k)的颜色为 $Q(i,j,k)$, $1\leqslant i,i'\leqslant m$, $1\leqslant j,j'\leqslant d$, $1\leqslant k\leqslant 2$。

1. 算法的基本原理

算法的基本策略是: 递归地将 n'个顶点 d'−路二分图分裂成两个 n'个顶点 $d'/2$−路二分图, 直到 $d'/2=2$ 时为止, 其中 d'是 2 的幂。然后利用 2−路图边着色算法对 $2m$ 个顶点 d−路二分图的边进行着色, 这里 n'的初值为 $2m$, d'的初值为 d。

设 $n=md$, 定义函数 γ_d: $\{1,2,\cdots,m\}\times\{1,2,\cdots,d\}\rightarrow\{1,2,\cdots,d\}$为:

$$\gamma_d(i,j)=d(i-1)+j,\quad 1\leqslant i\leqslant m, 1\leqslant j\leqslant d$$

同时令函数 α_d: $\{1,2,\cdots,n\}\rightarrow\{1,2,\cdots,m\}$及函数 $\beta_\alpha\{1,2,\cdots,n\}\rightarrow\{1,2,\cdots,d\}$的定义分别为:

$$\alpha_d(k)=\lfloor(k-1)/d\rfloor+1,\quad 1\leqslant k\leqslant n$$

$$\beta_d(k)=((k-1)\bmod d)+1, 1\leqslant k\leqslant n$$

2. 算法的形式化描述

下面的算法使用了 3 个二维数组 I', J'和 Q^*, 其中 I', J'存贮 2−路图, Q^*存贮 2−路图的边着色。

算法 12.9 EDGE COLORING d−WAY BIPARTITE GRAPHS

输入：2m 个顶点 d−路($d=2^s$)二分图的三维存贮矩阵 I，J；
输出：顶点的着色三维矩阵 Q。

```
procedure  Edge_Coloring_d_Way_Bipartite_Graphs  N(I,J,Q);
  begin
  (1)  for each i, j : (1≤ i≤ m) & (1≤ j≤ d) pardo     /* 初始化 */
          for k←1 to 2 do
              Q(i,j,k)←j ;
          endfor
   endfor;
  (2)  for s←1 to ⌈logd⌉ do
   (2.1)  for each i, j : (1≤ i≤ m) & (1≤ j≤ d)  pardo
     (2.1.1)  for k←1 to 2 do
                 q←Q(i, j, k);
                 i*←γd(i,γ2(βd/2(q),k));          j*←αd/2(q);
                 I'(i*,j*)←0;     J'(i*,j*)←0;
                 i'←I(i,j,k);     j'←J(i,j,k);
                 if i'>0  then q'←Q(i', j',3−k);
                                 I'(i*,j*)←γd(i',γ2(βd/2(q'), 3−k));
                                 J'(i*,j*)←αd/2(q')
                 endif
              endfor
           endfor;
   (2.2)  call Edge_Colring_2_Wag_Graph (I', J', Q*);
   (2.3)  for each i, j : 1≤ i≤ m  and  1≤ j≤ d pardo
             for k←1 to 2 do
                 q←Q(i, j, k);
                 i*←γd(i, γ2(βd/2(q), k));      j*←αd/2(q);
                 Q(i, j, k)←γ2(βd/2(q), Q*(i*, j*))
             endfor
           endfor
      endfor
  end.
```

算法 12.9 在执行 s 次循环后，(1) 顶点$(i,1,k)$, …, (i,d,k)的颜色 $Q(i,1,k)$, …, $Q(i,d,k)$都不相同；(2) 对每个分枝$((i,j,1),(i',j',2))$来讲，$Q(i,j,1)$与 $Q(i',j',2)$模 2^s 同余；(3) d−路图已分解成$(d/2^s)$−路子图。在同一子图中的所有顶点(i,j,k)都有相同的$\beta_{2^s}(Q(i,j,k))$值，d−路图的顶点(i,j,k)可被看作$(d/2^s)$−路子图的顶点$(i,(\alpha_{2^s}(Q(i,j,k)),k)$，其中$1\leq i,i'\leq m$，$1\leq j,j'\leq d$，$1\leq k\leq 2$。

定理 12.12 在 SIMD−EREW PRAM 上，对一个顶点最大度数为 d（d 是 2 的幂）的 n 个顶点的二分图进行边着色，算法 12.9 需 $O(\log d\log n)$时间、$O(nd)$处理器。

证明 根据算法 12.9，第(1)步需 $O(md)=O(nd)$处理器及 $O(1)$时间；第(2)步执行了 $O(\log d)$次，每次调用算法 12.8，需 $O(\log n)$时间，故整个第(2)步需 $O(\log d\log n)$时间和

$O(nd)$处理器。所以整个算法需 $O(\log d\log n)$时间、$O(nd)$处理器。

12.2.5 正整数 d–路二分图边着色的并行算法

上一小节介绍了 d 是 2 的幂次方的 d–路二分图的边着色算法，这里我们放宽这个限制，设 d 为任意正整数，我们来讨论 d–路二分图边着色并行算法，在 SIMD–EREW PRAM 上，算法需 $O(\log d\log^2 n)$时间、$O(nd)$处理器。下面就介绍这种并行算法。

1. 算法的基本原理

本节介绍的算法和前面的算法不同，这里采用了迭代改进策略，每次迭代使不一致的着色边数目减少一个常数倍；最终达到对所有边进行 d–着色的目的。算法的关键是：每次选取颜色为尽可能大的 2 次幂（但不超过 d）的子图而且用算法 12.8 进行着色。只要选择恰当，每次迭代时两端点着色不一致的边都减少了一个常数倍。

这种将任意的 d 归约为 d^* 是 2 的幂次方情形，最早是由 Gabow 等人建议的[11]。算法的一个重要子过程是：每次迭代至少选择了不一致边总数的 1／6 条边进行合法着色。

因为每次循环迭代时使得至少有 1／6 的不一致着色边进行合法着色，所以需经过 $\lceil\log_{\frac{6}{5}} n\rceil$次迭代才能使得整个图的边着色。算法分为两个阶段。第一阶段是选择枚举法，在不改变合法着色边颜色的条件下，对 d–路图的顶点集$\{(i,1,k), (i,2,k), \cdots (i,\mathrm{d},k)\}$枚举选择$\{1,2,\cdots,\mathrm{d}\}$中的值，使得至少有 1／6 不一致的着色分枝$((i,j,1), (i',j',2))$的端点颜色 $Q(i,j,1)$, $Q(i',j',2)$都在集合$\{1,2,\cdots,d^*\}$中，其中 $d^*=2^{\lfloor\log_2 d\rfloor}$ 。并令 $n^*=md^*$。第二阶段的工作是：顶点集$\{(i,1,k), \cdots, (i,d,k)\}$的着色是枚举选择$\{1,2,\cdots,d^*\}$的值，使得端点颜色属于$\{1,2,\cdots,d^*\}$的分枝都得到的合法边着色。但不改变端点颜色在$\{1,2,\cdots,d^*\}$的分枝都得到的合法边着色。这样，原来不一致的着色边经过一次循环迭代后，至多还有 5／6 的分枝仍为不一致着色边。

设集合$\{1,2,\cdots,d\}$划分为四个子区间 $D_1=\{a_1+1,\cdots,a_1+l_1\}$，$D_2=\{a_2+1,\cdots,a_2+l_2\}$，$D_3=\{a_3+1,\cdots,a_3+l_3\}$，$D_4=\{a_4+1,\cdots,a_4+l_4\}$，其中：$a_1=0$，$l_1=\lfloor(d+3)/4\rfloor$ ，$a_2=a_1+l_1$，$l_2=\lfloor(d+2)/4\rfloor$ ，$a_3=a_2+l_2$，$l_3=\lfloor(d+1)/4\rfloor$ ，$a_4=a_3+l_3$，和 $l_4=\lfloor d/4\rfloor$ 。

令 $C_1=D_1\cup D_2$，$C_2=D_1\cup D_3$，$C_3=D_1\cup D_4$，$C_4=D_2\cup D_3$，$C_5=D_2\cup D_4$ 和 $C_6=D_3\cup D_4$，对 $r\in\{1,2,\cdots,6\}$，令 χ_r 是 C_r 的特征函数，定义如下：

$$\chi_r(j)=\begin{cases}1, & j\in C_r\\ 0, & j\notin C_r\end{cases}$$

对任何属于 $\{1,2,\cdots,d\}$ 的一对颜色 q 及 q'，存在这样一个 $r\in\{1,2,\cdots,6\}$，使得 q 及 q' 都属于 C_r。这样，至少有 1／6 的不一致着色边，它的端点颜色都属于 C_r。

若 $\{x+1,\cdots,x+t\}$ 和 $\{y+1,\cdots,y+t\}$ 是 $\{1,2,\cdots,d\}$ 的两个不相交的子区间。令 $\tau_{x,y,t}$ 是交换这两个区间集合 $\{1,2,\cdots,d\}$ 的一个枚举。

$$\tau_{x,y,t}(j)=\begin{cases} j-x+y, & 若 x+1\leqslant j\leqslant x+t \\ j+x-y, & 若 y+1\leqslant j\leqslant y+t \\ j, & 否则 \end{cases}$$

令 π_1 是 $\{1,2,\cdots,d\}$ 的单位枚举，$\pi_2=\tau_{a_2,a_3,t_3}$，$\pi_3=\tau_{a_2,a_4,t_4}$，$\pi_4=\tau_{a_1,a_3,t_3}$，$\pi_5=\tau_{a_1,a_4,t_4}$ 和 $\pi_6=\tau_{a_1,a_3,t_3+t_4}$，那么对每个 $r\in\{1,2,\cdots,6\}$，π_r 将 C_r 映射到 C_1，$C_1\in\{1,2,\cdots,d^*\}$。

2. 算法的形式化描述

算法 12.10 EDGE–COLORING OF BIPARTITE GRAPHS WITH DEGREE d

输入：$2m$ 个顶点 d–路二分图的三维存贮矩阵 I，J；

输出：顶点着色的三维存贮矩阵 Q。

```
procedure Edge_Coloring_Bipartite_Graphs(G);
 begin
  (1)  for each i, j : (1 ≤ i ≤ m) and (1 ≤ j ≤ d)  prado        / * 初始化 * /
           for k←1  to  2  do
             Q(i,j,k)←j
           endfor
       endfor;
  (2)  for s←1 to ⌈log_{6/5}n⌉ do    / * 第一阶段 * /
    (2.1)   a←−1;
    (2.2)   for  r'←1  to  6  do
      (2.2.1)   for each i,j : (1 ≤ i ≤ m)  and  (1 ≤ j ≤ d)   pardo
                    A(i,j )←0;     / * 作为工作数组 * /
                    i'←I(i,j ,1);       j '←J(i,j ,1);
                    if i' > 0   then  if  Q(i,j ,1)=Q(i',j ',2)    / * 此分枝合法着色 * /
                                       then  A(i,j )← χ_r(Q(i,j,1)) * x_r(Q(i',j ',2))
                          / * 计算此分枝端点是否属于集合 C_r * /
                                  endif
                    endif
                endfor;
      (2.2.2)   for each i,j :  (1 ≤ i ≤ m)   and   (1 ≤ j ≤ d)   pardo
                    a'← Σ_{i=1}^{m} Σ_{j=1}^{d} A(i,j)
                endfor;
      (2.2.3)   if  a < a' then  a←a';   r←r'       endif
                   / * 分枝端点的颜色在集合 C_r 中且含分枝数目最多的是 C_r * /
          endfor;
          / * 应用 π_r 对着 C_r 内颜色的分枝着色 * /
    (2.3)   for each i,j : (1 ≤ i ≤ m)   and  (1 ≤ j ≤ d)   pardo
              for  k←1   to  2  do
                  Q(i,  j,  k)←π_r(Q(i,j,k))
              endfor
          endfor;
```

```
(2.4) for each i,j : (1≤i≤m) and (1≤j≤d) pardo      /* 第二阶段 */
        for k←1 to 2 do
          q←Q(i,j,k);
          if q≤d*   /* 顶点着色属于{1,2,…,d*} */
            then I1(i,q,k)←0;   J1(i,q,k)←0;      /* 工作数组 */
                 i'←I(i,j,k);     j'←I(i,j,k);
                 if i'>0   /* 若为真，表示存在一条分枝 */
                   then q'←Q(i',j',3-k);      /* 此分枝另一端点颜色 */
                        if q'≤d*  then I1(i,q,k)←i';  J1(i,q,k)←q'  endif
                 endif
          endif
        endfor
      endfor;
(2.5) call Edge_Coloring_d_way_Bipartite_Graphs  N(I1,J1,Q*);
   /* 对一些分枝重新着色，使其成为合法着色边 */
(2.6) for each i,j : (1≤i≤m) and (1≤j≤d) pardo
        for k←1 to 2 do
            q←Q(i,j,k);
            if q≤d*  then Q(i,j,k)←Q*(i,j,k)     endif
        endfor
      endfor
   endfor
end.
```

算法 12.10 在执行了 s 次迭代后，(1) 顶点$(i,1,k),\cdots,(i,d,k)$的颜色 $Q(i,1,k),\cdots,Q(i,d,k)$ 都不相同；(2) 至多有$(5/6)^s n$ 条分枝$((i,j,1),(i',j',2))$，它们的顶点颜色 $Q(i,j,1)$及$Q(i',j',2)$不一致，其中$1\leq i,\ i'\leq m$，$1\leq j,\ j'\leq d$，$1\leq k\leq 2$。

定理 12.13 在 SIMD–EREW PRAM 上，对顶点最大度数为 d（d 为任意正整数）的 n 个顶点的二分图进行边着色，算法 12.10 需 $O(\log d\log^2 n)$时间、$O(nd)$处理器。

证明 根据算法 12.10 可知，第(1)步需 $O(1)$时间和 $O(nd)$处理器；第(2)步需执行 $O(\log n)$次。第(2.1)步需 $O(1)$时间及处理器；第(2.2)步需 $O(\log n)$时间（求和）及 $O(nd)$处理器；第(2.3)步需 $O(1)$时间及 $O(nd)$处理器；第(2.4)步需 $O(1)$时间及 $O(nd)$处理器；第(2.5)步由定理 12.12 可知，需 $O(\log d^* \log n^*)$时间及 $O(n^* d^*)$处理器；第(2.6)步需 $O(1)$时间、$O(nd)$处理器。因此，第(2)步共需 $O(\log d\log^2 n)$时间$(d^*=2^{\lfloor\log d\rfloor}\leq d,\ n^*=md^*\leq nd)$和 $O(1)$处理器。故整个算法需 $O(\log d\log^2 n)$时间和 $O(n)$处理器。若 $d=O(n)$，则算法需 $O(\log^3 n)$时间。

12.2.6 多重图边着色的并行算法

对二分多重图来讲，我们从上节可知，其边着色算法的基本思想：首先找出图的欧拉划分，然后再对边进行着色。假设 G 是一个要着色的图，且它的最大度数为偶数，增加一个新顶点且把这个新顶点与 G 的奇数度顶点相连，增值 G 成为一个欧拉图 G^+。找出G^+ 欧拉

回路，任选一个顶点 v，令 e_i 是从 v 出发遍历的第 i 条边，那么 G 的边被划分为两个集合，一条边属于哪个集合依赖于 e_i 的下标 i 是奇数还是偶数。这样每个集合导出的子图最大度数为 $\Delta/2$，如此递归地分裂，直至导出简单回路或路径时为止，这时就是 2−路图。利用上节介绍的技术，由定理 12.13，我们可以在 $O(\log^3(\Delta n))$ 时间内利用 $O(\Delta n)$ 处理器给一个二分多重图的边进行并行着色，其中 n 是 G 的顶点数目，Δ 是图 G 的顶点最大度数。

1. 算法的基本原理

我们假定图 $G(V,E)$ 是一个连通的多重图，不然的话，利用第四章的顶点倒塌法可以求出每个连通分支，然后同时对每个连通分支进行边着色。对一般多重图，构造一个有效的并行算法仍要利用欧拉遍历技术。Karloff 等人[12]基于 SIMD−CRCW PRAM，介绍了多重图 G（V,E） 着 3 $\lceil\Delta/2\rceil$ 种颜色的并行算法。算法工作过程是：首先将 G 转换成一个辅图 G_B，G_B 是一个二分多重图，对 G_B 边并行着色；然后令 M_i 表示 G_B 中着颜色 c_i 的边的集合，显然 M_i 是 G_B 的一个匹配，位于 M_i 这些边对应多重图 G 的边导出一个子图 G_i，G_i 的最大度数小于等于 2，即 G_i 是路径或简单回路的集合；最后，对每个 G_i 用 3 种颜色进行着色。因为 G 至多分解成（$\Delta/2$）个子图，故共需 3 $\lceil\Delta/2\rceil$ 种颜色对 G 的边进行着色，$1\leqslant i\leqslant\lceil\Delta/2\rceil$。

2. 算法的非形式化描述

算法 12.11 EDGE−COLORING OF MULTIGRAPHS;

输入：多重图 $G(V,E)$ 的边集合；

输出：每条边 e 着色 $c(e)$，$e\in E$。

procedure Edge__Coloring__of__Multigraph (V,E); / * G(V,E)是一个多重图 * /

begin

(1) 构造一个二分多重图 $G_B(X,Y,E_B)$ 如下：

(1.1) 若 G 不是一个欧拉图，增加一个顶点 v 到 G 中，且增加 z 与所有奇数度顶点关联的边，形成欧拉图 G';

(1.2) 应用第九章算法 9.1 找出 G' 的有向欧拉回路；

(1.3) 对 $v\in V$，令 $v'\in X,\bar{v}\in Y$； $E_B\leftarrow\phi$;

(1.4) 从有向欧拉回路某一顶点出发遍历欧拉回路，若遍历次序是从 v 到 u，则加一条无向边 $(v,\bar{u})$ 到 E_B 中；

/ * 构造出的 G_B 的最大度数至多为 $\lceil\Delta/2\rceil$ * /

(2) **call** Edge__Coloring__Bipartite__Graphs(G_B); / * 调用算法 12.10 * /

(3) 令 M_i 是 G_B 中着颜色 c_i 的边集合，对每个 c_i，构造一个多重图 G_i 如下：

对每条边 $(u',\bar{v})\in M_i$，将边 (u,v) 加入 E_i 中，因 M_i 是 G_B 的一个匹配，对应多重图 G_i 的度数 $\leqslant 2$，即 G_i 是路径和简单回路的集合；

(4) 每个 G_i 同时用三种颜色着色，即调用算法 12.8，若检测到奇数回路，则将一条边着第三种颜色，其余边用另两种颜色着色

end.

定理 12.14 在 SIMD−CRCW PRAM 上，对一个多重图 $G(V,E)$，$|V|=n$ 的边用 3

$\lceil\Delta/2\rceil$种颜色并行地着色，算法 12.11 需 $O(\log^3(\Delta n))$时间和 $O(\Delta n)$处理器，这里 Δ 是 G 的顶点最大度数.

证明 根据算法 12.11，第(1.1)步需 $O(\log n)$时间、$O(n\Delta)$处理器；第(1.2)步由定理 9.1 可知，需 $O(\log(n\Delta))$时间和 $O(n\Delta)$处理器；第(1.3)步需 $O(1)$时间和 $O(n)$处理器；第(1.4)步需 $O(\log(n\Delta))$时间和 $O(n\Delta)$处理器。故第(1)步需 $O(\log(n\Delta))$时间和 $O(n\Delta)$处理器。第(2)步由定理 12.13，需 $O(\log^3(n\Delta))$时间和 $O(n\Delta)$处理器．第(3)步需 $O(1)$时间、$O(n\Delta)$处理器。第(4)步由定理 12.11 知，需 $O(\log n)$时间和 $O(n\Delta)$处理器．因此，整个算法需 $O(\log^3(n\Delta))$时间、$O(n\Delta)$处理器．

Karloff 等人基于同一计算模型，还考虑 $\Delta=3$ 这一特殊情况下多重图边着色的并行算法，他们给出了一个 $O(\log n)$时间和 $O(n+m)$处理器的算法[12]．

12.3 小 结

对图论上一个比较复杂而又具有广泛应用背景的着色问题，本章重点叙述了这一问题的并行算法．主要内容包括：顶点度数为常量的图的顶点着色算法及其在计算图的极大独立集中的应用；平面图顶点的 7–着色和 5–着色并行算法以及最优的 5–着色并行算法；树的边着色快速并行算法；二分图的边着色并行算法以及多重图边着色的有效并行算法等。

八十年代末期，国际上对图的着色问题的并行化进行了较深入的研究，取得了丰硕的成果。除了我们介绍的一些典型算法外，还有许多重要文献成果尚待引人．Karchmer 等人[13]基于 SIMD–CRCW PRAM，对不含奇数边回路的图用 Δ 种颜色对图顶点进行着色，给出了一个 $O(\log^3 n)$时间、$O(n+m)$处理器的并行算法；而 Boyar 等人[14]基于 SIMD–CRCW PRAM，对平面图顶点问题，建议了一个 5–着色随机并行算法，他们算法期望的时间复杂性为 $O(\log^3 n)$，使用了 $O(n^2)$个处理器；有关图的边着色的并行算法，Karloff 等人[12]基于 SIMD–CRCW PRAM，对著名的 Vizing 定理进行并行化，给出了一般简单图用$(\Delta+1)$边着色的并行算法，他们的算法需 $O(\Delta^{O(1)}\log^{O(1)}n)$时间和 $O(n^{O(1)})$处理器；显然仅当 $\Delta=O(\log^{O(1)}n)$ 时此算法方为有效，随后基于同一计算模型，他们又建议了此问题的一个并行随机算法，算法的期望时间为 $O(\log^{O(1)}n)$和 $O(n^{O(1)})$处理器．后来，又有人对平面图边着色问题进行了研究，提出了平面图的边着色并行算法，如 Chrobak 等人[15]基于 SIMD–EREW PRAM，给出了一个 $O(\log^2 n)$时间，使用 $O(n)$处理器的算法，其中着色数为 $\max\{\Delta,19\}$，随后 Chrobak 等人[16]又改进了他们的算法，基于 SIMD–EREW PRAM，提出了一个 $O(\log^3 n)$时间和 $O(n)$处理器的并行算法，其中着色数为 $\max\{\Delta,8\}$；Boyar 等人[14]基于 SIMD–CRCW PRAM，给出了一个 $O(\log^3 n)$时间、$O(n^2)$处理器的算法，使用的颜色数为 $\Delta\geqslant 23$．Gibbons 等人[9]基于 SIMD–CREW PRAM 还给出了一种特殊的图(Halin 图)的边着色并行算法，他们的算法需 $O(\log n)$时间及 $O(n)$处理器。

参 考 文 献

[1] Goldberg A V, Plotkin S A. Parallel (Δ+1)–Coloring of Constant –Degree Graphs, *Inform. Proc. Lett.*, **25**,1987, 241–245

[2] Goldberg A V, Plotkin S A, Shannon G. Parallel Symmetry–Breaking in Sparse Graphs, *Proc. 19th Annu. ACM Sympo. on Theory of Computing*, 1987, 315–324

[3] Naor J. A Fast Parallel Coloring of Planar Graphs with Five Colors, *Inform. Proc. Lett.*, **25**, 1987, 51–53

[4] Hagerup T, Chrobak M, Diks K. Optimal Parallel 5–Coloring of Planar Graphs. *SIAM J. Comput.*, **18**(2), 1989, 288–300

[5] Cole R, Vishkin U. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking, *Inform. and Control*, **70**, 1986, 32–53

[6] Cole R, Vishkin U. Appoximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time, *SIAM J. Comput.*, **17**(1), 1988,128–142

[7] Vizing V G. On An Estimate of the Chromatic Class of a p–Graph, (*in Russinan*), *Diskret. Anal*, 3, 1964, 25–30

[8] Shannon C E. A Theorem on Coloring Lines of a Network, *J. Math. Phys.*, **28**, 1949, 148–151

[9] Gibbons A M, Israeli A, Rytter W. Parallel O(log*n*) Time Edge–Coloring of Trees and Halin Graphs, *Inform. Proc. Lett.*, **27**, *1988*,43–51

[10] Lev G F, Pippenger N, Valiant L G. A Fast Parallel Algorithm for Routing in Permutation Networks, *IEEE Trans. Comput.*, C–**30**(2), 1981, 93–100

[11] Gabow H N, Kariv O. Algorithms for Edge Coloring Bipartite Graphs and Multigraphs, *SIAM J. Comput.*, **11**, 1982, 117–129

[12] Karloff H J, Shmoys D B. Efficient Parallel Algorithms for Edge Coloring Problems, *J. Algorithms*, **8**, 1987, 39–52

[13] Karchmer M, Naor J. A Fast Parallel Algorithm to Color a Graph with Δ Colors, *J. Algorithms*, **9**, 1988, 83–91

[14] Boyar J F, Karloff H J. Coloring Planar Graphs in Parallel, *J. Algorithms*, **8**, 1987, 470–479

[15] Chrobak M, Yung M. Fast Algorithms for Edge–Coloring Planar Graphs, *J. Algorithms*, **10**,1989, 35–51

[16] Chrobak M, Nishizeki T. Improved Edge–Coloring Algorithms for Planar Graphs, *J. Algorithms*, **11**, 1990, 102–116

第十三章　组　合　搜　索

在这一章，我们讨论用状态空间图表示的组合搜索问题．所谓**组合搜索**(Combinatorial Search)，就是找"定义问题空间中的一个或多个最优解或次优解"的过程[1]．在离散的、有限的数学结构上执行计算，就是组合算法(Combinatorial Algorithm)[2]．组合搜索问题在人工智能、运筹学等应用领域中经常出现，例如定理证明、博奕比赛、找图的最短路径、求旅行商的周游路线等．为了简化，这里只讨论树表示的搜索．

组合搜索问题可以分为组合判定问题和组合优化问题．前者是找满足给定约束条件的解，后者是求最优解，即寻找一个满足给定的约束条件并使其目标函数值达到最大或最小的解．在人工智能中的应用问题，有些属于组合优化问题，如图象识别与声音识别中的最佳匹配；有些属予组合判定问题，例如定理证明和某些诊断、决策型专家系统．

一个搜索问题可以表示为一棵树，称作**状态空间树**(State Space Tree)．树的根结点代表要被求解的原始问题，非叶结点（内部结点）代表子问题．除叶结点外，所有结点可以区分为"**与结点**"和"**或结点**"两类．与结点代表的子问题，只有当它的全部孩子结点都得到解答时它才能解决．相反，对于一个或结点，只要它的一个孩子结点已经有了解答，这个或结点代表的子问题就可以解决．如果一棵状态树仅仅包含与结点，则称之为"**与树**"，如图 13.1(a)所示．实现与树搜索的方法是分治算法，因为问题的解是由它的全部子问题的解组合而成的．如果一棵状态空间树只包含或结点，则称它为"**或树**"，如图 13.1(b)所示．实现或树搜索的常用方法是分枝限界算法，因为问题的解可由它的任何一个子问题的解而得到．如果一棵树既包含与结点又包含或结点，则称其为"**与或树**"，如图 13.1(c)所示．**博奕树**(Game Trees)是与或树的一个例子，它的搜索是运用**修剪**(Pruning)技术来减少计算量，通常叫做 α-β 搜索算法．

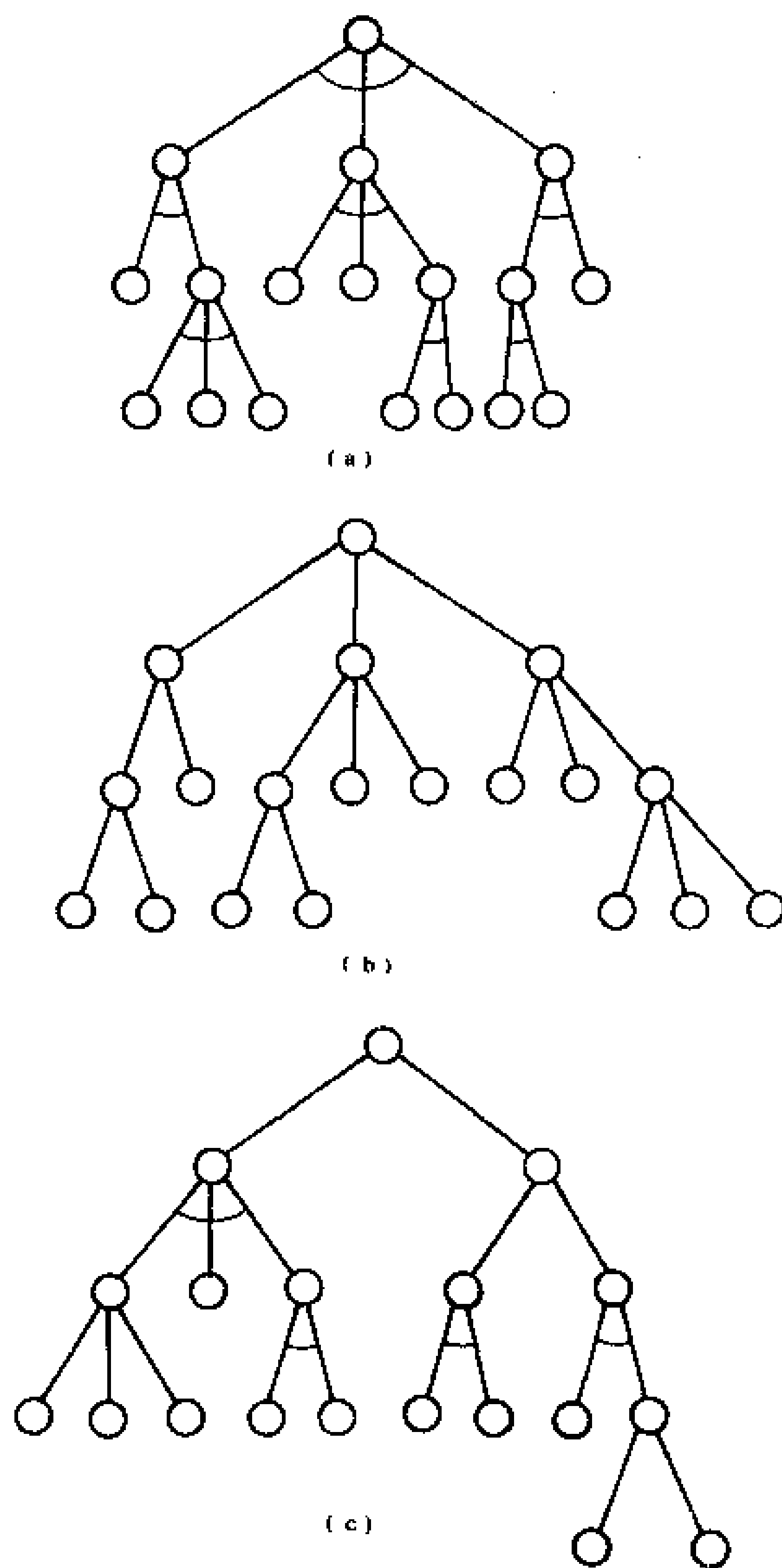


图 13.1　用树表示搜索问题

(a) 与树；(b) 或树；(c) 与／或树

在 13.1 节，我们将讨论与树的并行搜索，它是分治方法的典型应用。13.2 节比较详细地介绍并行分枝限界算法，并且证明并行分枝限界算法存在着异常现象，有的情形是增加处理器数目可能增加搜索的执行时间，有的场合是增加处理器的数目则可能导致执行时间不成比例的减少。$\alpha-\beta$ 搜索算法将在 13.3 节出现，若干并行化的 $\alpha-\beta$ 搜索方法将要进行讨论。

13.1 分治法(Divide and Conquer)

分治方法是一种问题求解的方法学，求解的思想就是将整个问题分成若干个特性相同的小问题后分而治之。通常，由分治方法所得到的子问题与原问题具有相同的类型。若得到的子问题相对来说还是太大，则可反复使用分治策略将这些子问题分成更小的子问题，直至产生不用进一步细分就能求解的子问题为止。也就是说，分治方法是递归的，可以用递归过程来表示。

13.1.1 SIMD 模型的分治算法

在 SIMD 模型上，使用分治策略设计的并行算法描述如下[3]:

算法 13.1 PARALLEL DIVIDE-AND-CONQUER ALGORITHM

```
procedure D&C (I: structure; O: structure);
        /*过程 D&C 的输入和输出分别在结构 I 和 O 中*/
  Local S1,S2,…,Sk: structure;   Local T1,T2,…,Tk: structure;
  begin
    if SMALL(I, O)
      then return (ANSWER(I;O));
      else /*将问题分成 k 个类型相同的子问题，由输入结构 I 形成 k 个子
             结构 S1,S2,…,Sk.对于 k 次递归调用的输出，指定 k 个临时
             结构 T1,T2,…,Tk.*/
      SPLITINPUT(I:S1,S2,…,Sk);
         /*并行执行 k 次递归调用 D&C*/
      cobegin
             call D&C(S1;T1); call D&C(S2;T2); … ;call  D&C(Sk;Tk);
      coend;
      COMBINE(T1,T2,…,Tk,I:O)
    endif
  end.
```

其中：**cobegin–coend** 表示同时执行 k 次递归调用 D&C。SMALL 是一个布尔函数，当问题的规模足够小、能够直接计算解答时，SMALL(I,O)的值为“真”，则返回由 ANSWER 直接计算出来的结果；否则，SMALL(I,O)的值为“假”，SPLITINPUT 从

原来的输入结构 I 提供 k 个输入结构，并行执行 k 个递归调用 D&C 的每一个调用，接收这些输入结构中的一个，并且对它的输出指定一个存贮区．k 个递归调用 D&C 并行地执行完成后，COMBINE 将子问题的解 T_1，T_2，…，T_k 及原来输入结构 I 的某些可能的值，产生原来问题的解答．在这里，**structure** 表示一个或多个更一般的数据类型。

在本章的开始我们曾提到，一个问题的分治解法，可以用一棵与树来表示，这是因为用内部结点表示的任何问题，它的解需要全部子问题的解，而这些子问题是由该结点的孩子结点表示的．因此，分治算法亦可视作自底向上搜索"与树"的过程，而且整棵树中的每一个结点都必须被计算．由于递归调用 D&C 产生的 k 个子问题可以同时求解，所以分治算法自然地体现了并行执行的概念，可以并行地自底向上搜索一棵与树．

13.1.2 分治法在 MIMD 模型上的实现途径

在 MIMD 计算机上实现分治算法，已经提出三种途径．第一种方法是多处理器按树形连接，即建造一棵和搜索树相对应的处理器树．图 13.2 给出一棵满的三元处理器树，它是 k-元处理器树的一个例子．其中：P_i 表示处理器，M_i 表示局部存储器，树边表示处理器之间的连接线路．这种简单连接的树机器适合于 VLSI 实现．然而，这种方法有两点不足：其一是根处理器将成为系统的瓶颈口，因为它是所有输入和输出的通道；其二是具有固定连接结构的处理器树，没有足够的灵活性，但它只对某些分治算法是合适的．

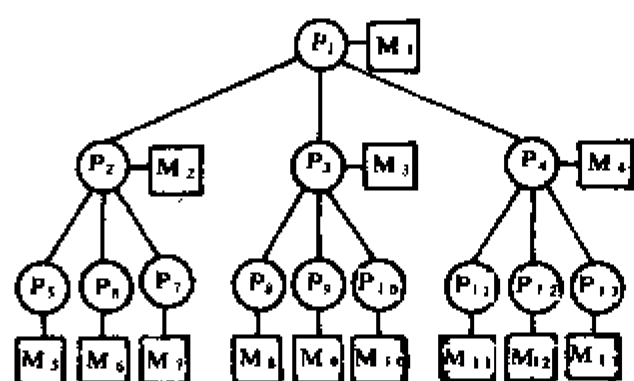


图 13.2 三元多处理器树

第二种方法是使用虚拟的机器[4]．它由若干个处理器组成，每个处理器带有自己私有的存储器，处理器采用健全的互连网络相连，例如二进制的超立方连接．它具有相适应的算法决定子问题应在何时何处求解．因为分治算法进程通讯的层次性，使得子问题可以有效地映射到这种体系结构上．图 13.3 所示的体系结构是 $n=3$ 的立方连接．

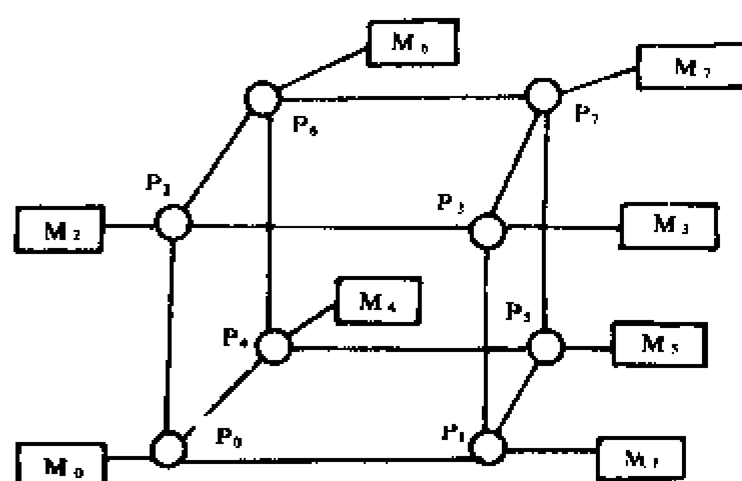


图 13.3 3 立方连接的虚拟树机

第三种方法建议使用紧耦合多处理器执行分治算法，可看作是上述两种方法的变种．它是基于共享存贮的总线结构，即所有的处理器通过公共总线连接到共享存贮器．由于数据在执行过程中必须共享，存贮器或总线可能变成一个瓶颈口．为了达到足够高的共享存贮和总线带宽，存贮器可以划分成独立的模块，总线可以由几条总线组成，其中每一条都连接到所有的处理器和若干个存贮器模块上，如图 13.4 所示．

13.1.3 分治算法的复杂性[1]

由此看来，为了计算一棵与树，其职能要求是处理器互连的密集度(Conglomerate)，

而重要的是确定并行化的粒度(Granularity)。所谓并行化的粒度，指的是一个处理器为了达到最优性能所计算子问题的最小规模。粒度反映着处理器互连的密集情况。若粒度大，则处理器是松耦合的；否则是紧耦合的。用什么标准来衡量粒度的大小呢？通常使用的标准是**处理器利用**(Processor Utilization，简记作 PU)，kT^2 或 AT^2。其中：k 是处理器的数目，T 是计算时间，而 A 是 VLSI 实现的面积。

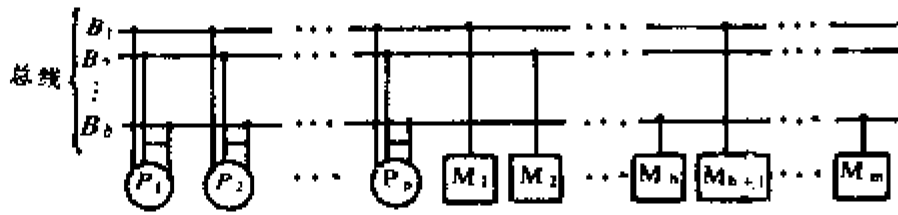


图 13.4 多总线、多模块存贮器的多处理器

在 SIMD 模型上研究分治算法的复杂性时，我们确信它满足最优的 PU 条件。一棵与树的并行搜索。粗略地划分成三个阶段：**启动**、**计算**和**组合求解**。在启动阶段，问题被划分且任务传播到网络，大多数启动阶段，任务少但处理器多，多数处理器处于空转，直至它们被给定一个问题要划分和传播为止。在计算阶段中，所有的处理器保持繁忙状态，直到系统中的任务个数小于处理器个数为止。在组合求解阶段，将子问题的解答组合起来产生原问题的解。在这个阶段。又存在任务比处理器少的现象，某些处理器忙于组合结果，而其它的处理器则是空转。因此，并行搜索与树的处理器利用 PU 依赖于消耗在计算阶段的时间总数与其它阶段的时间总数之比值。若比值大，则 PU 大。为了达到 PU 大，要求在启动阶段和组合求解阶段的处理器空转时间少。

对于一棵 n 层的二元与树的并行搜索，其时间复杂性可以用下列递归方程表示：

$$T(n)=\begin{cases}S(n)+2T(n/2)+C(n), & n>1\\ O(1), & \text{否则}\end{cases}$$

其中：$S(n)$ 和 $C(n)$ 是启动和组合求解阶段的时间复杂性。

由此可见，要得到最优处理器利用的粒度，与 $S(n)$和 $C(n)$的复杂性有关。例如，在求 n 个数的和或者求它们的最大值问题中，使用 $n/(\log n)$个处理器，$S(n)+C(n)=O(1)$，处理器达到最大的利用，即 PU 是最大。在排序 n 个数的问题中，使用 $\log n$ 个处理器，$S(n)+C(n)=O(n)$，处理器利用达到最大。关于渐近的处理器利用，当 $S(n)+C(n)=O(n)$ 时，$n/\log n$ 是一个门限值。对于 k(n 的一个函数)个处理器而言，处理器利用 PU 具有下述性质[21]：

$$\text{当 } \lim\frac{k}{n(\log n)}=0 \text{ 时，则 } \mathrm{PU}=1$$

$$\text{当 } \lim\frac{k}{n(\log n)}>0 \text{ 时，则 } 0<\mathrm{PU}<1$$

$$\text{当 } \lim\frac{k}{n(\log n)}=\infty \text{ 时，则 } \mathrm{PU}=0$$

因为处理器利用随着处理器数目的减少而增加，它不是一个并行处理器效率的足够的尺度。更为合适的尺度是 kT^2 准则，这个准则既考虑处理器利用又考虑计算的时间。在并行分治算法中，使得 kT^2 最小的渐近最优的处理器数目，当 $S(n)+C(n)=O(1)$时，它是 $O(n/\log n)$。模拟计算的结果，证实了最优的处理器数目或者精确地是 $n/(\log n-1)$，或者非常接近这个值。

最后还要指出：最优的粒度依赖于问题的复杂性、与树的形状（与树是否平衡）、以

及沿着每一条路径上各结点计算时间的分布[5]。

13.2　分枝限界法(Branch and Bound)

求解组合搜索问题的常用方法是分枝限界算法[6,7]，它是回溯法(Backtracking)的变种，是一种流行的算法设计技术。分枝限界法利用部分解的最优性信息，避免考虑那些不能导致最优的解，从而快速地求得问题的解答。

13.2.1　8一谜问题——一个例子

现在，我们先用一个实例来介绍分枝限界方法。Sam Loyd 1878 年提出了 15-谜问题，这里考虑简化的 8-谜问题，如图 13.5 所示。在一个 3×3 的棋盘上排放着编号为 1 至 8 的 8 张牌，8 个位置上恰好一个位置放一张牌，剩下一个位置为空格。对于任意给定

1	5	2
4	3	
7	8	6

(a) 初始排列；

1	2	3
4	5	6
7	8	

(b) 目标排列

图 13.5　8-谜问题

的一种初始排列，如图 13.5(a)所示的排列，要求通过一系列的合法的移动移成图 13.5(b)所示的排列。所谓合法的移动就是将邻近空格的上、下、左、右位置中的某一张牌移到空格内。对图 13.5(a)的情况，开始有三种可能的合法的移动，即将牌 2、3、6 之中的一张牌移到空格内。在做了这次移动之后，才可以做下一次移动。每移动一次，产生一种新的排列。每一种排列称作这个谜问题的一个**状态**(State)。初始排列和目标排列叫做初始状态和目标状态（答案状态）。若由初始状态到某个状态存在一系列的合法的移动，则称该状态可由初始状态到达。一个初始状态的**状态空间**(State Space)由所有可以从初始状态到达的状态组成。可以将此状态空间构造成一棵树，叫做**状态空间树**。初始状态是这棵树的根，所有可以从初始状态到达的状态是这棵树中的结点，每一条树边表示一次合法的移动(即是表示一次操作)。不难看出，移动牌和移动空格实质上是等效的，而且在做实际的移动时更加直观，因此我们以后都将从父状态到子状态的一次转换，看成是空格的一次合法的移动。对图 13.5(a)所示的初始状态，图 13.6 给出了这棵状态空间树的一部分。

对于任何一个问题，一旦设想出一种状态空间树，那么就可以通过系统地生成问题状态，接着确定这些问题状态中的哪些状态是解状态，最后确定哪些解状态是答案状态而将这问题解出。系统地生成问题状态，首先从根结点开始，然后生成其它的结点。如果已生成一个结点而它的所有孩子结点还没有全部生成，则这个结点叫做**活结点**。当前正在生成

其孩子结点的活结点叫做 **E-结点**（正在扩展的结点）．不再进一步扩展或者其孩子结点已全部生成的结点就是**死结点**（被抛弃的结点）．为了生成其它的结点，需要一张活结点表来保存活结点．若活结点表采用栈结构，则是深度优先生成法．若活结点表采用队列结构，则是宽度优先生成法．这两种生成法都是呆板而盲目的，它们不考虑开始的状态如何，总是按千篇一律的顺序生成（或搜索）、耗费许多时间在那些不能很快找到答案结点、甚至找不到答案结点的子树上搜索．

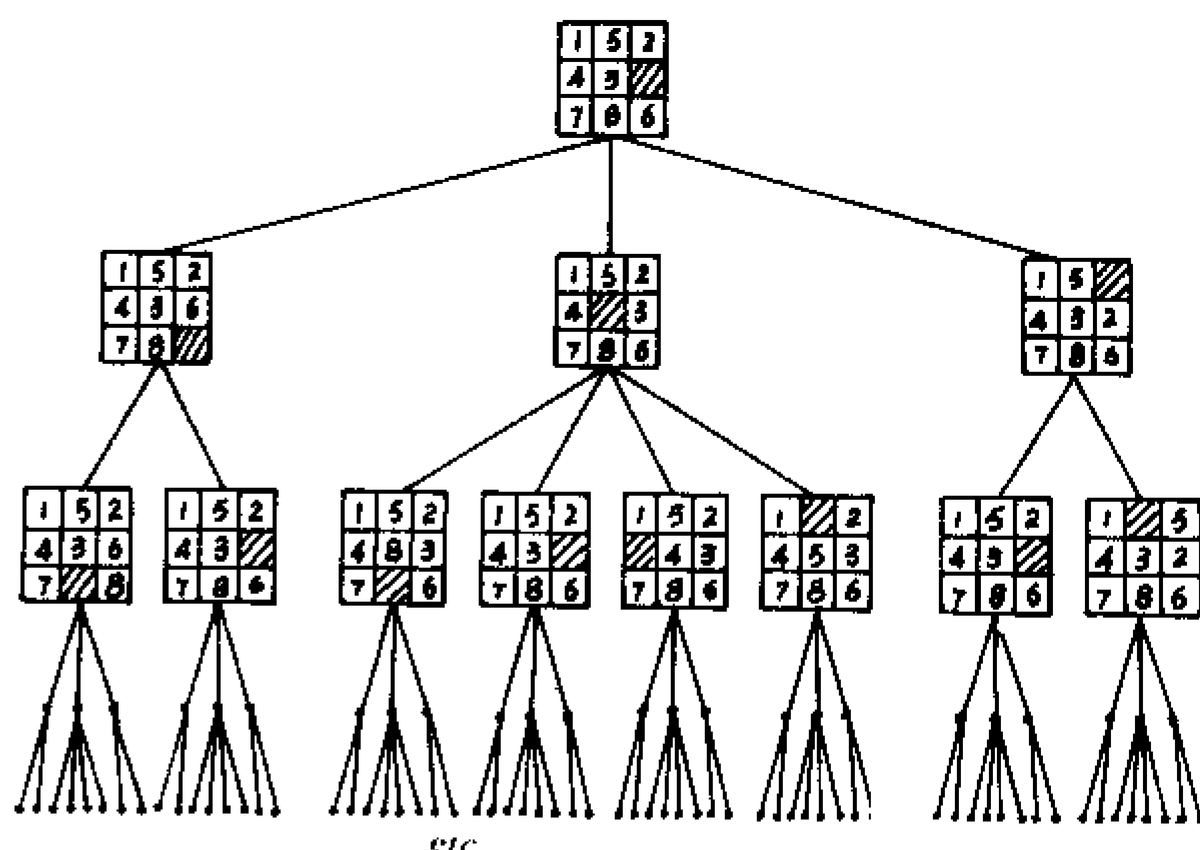


图 13.6 8-谜问题实例的部分状态空间树

如果对每个活结点定义一个"有智能的"的优先级函数 $f(\cdot)$，则按 $f(\cdot)$的大小选取下一个 E-结点往往可以加快到达一个答案结点的速度．$f(\cdot)$叫做结点的**代价函数**．定义如下：若 x 是答案结点，则 $f(x)$是由状态空间树的根结点到 x 的代价；若 x 不是答案结点且子树 x 不包含任何答案结点，则 $f(x)=\infty$，否则 $f(x)$等于子树 x 中具有最小代价的答案结点的代价。但要指出的是，要得到代价函数 $f(\cdot)$所用的计算工作量与解原问题具有相同的复杂度，这是因为计算一个结点的代价通常要搜索包含一个答案结点的子树 x 才能确定，而这正是解决此问题所要做的搜索工作，因此要得到精确的代价函数一般是不现实的．

切合实际的作法是给出一个便于计算代价估计值的函数 $g(x)=f(h(x))+\hat{g}(x)$，其中$\hat{g}(x)$是由 x 到达一个答案结点所需做的附加工作的估计函数；$f(\cdot)\neq 0$ 是一个非降函数。若 $f(\cdot)=0$，则可能会导致偏向于作纵深检查．使用代价估计值函数 $g(x)$选择下一个 E-结点的搜索策略总是选取 $g(\cdot)$值最小的活结点作为下一个 E-结点．称之为**最小代价搜索**(Least-Cost Search)，简称 **LC-搜索**，或者叫做**最好优先搜索**(Best-First Search)．可以选用堆结构(**min** 堆)来保存活结点表．

在求解 8-谜问题中，给出任意一个初始的状态，人们总是企图用尽可能少的合法移

动次数达到目标状态①。我们令 $f(h(x))$为从根结点到结点 x 的移动次数；令 $d(x)$为 x 所代表的状态中，所有错位的牌和它的正确位置之间的曼哈坦距离(Manhattan Distance)。注意：(x_1,y_1)与(x_2,y_2)的曼哈坦距离为$|x_1-y_1|+|x_2-y_2|$。使用这样的估计函数 $g(x)=f(h(x))+d(x)$，使得我们能尽可能集中地搜索那些能较快达到目标状态的子树。任何时候都从具有最小函数值的结点开始搜索，如果两个结点具有相同的函数值，那么先检测离根结点较远的结点。如果离根结点距离相同的两个结点具有相同的函数值，那么选择其中的任一结点搜索。搜索过程如图 13.7 所示。结点旁边园圈内的数字为 $g(\cdot)$的值。和宽度优先搜索法相比，LC–搜索法所需检测的结点数目少得多。

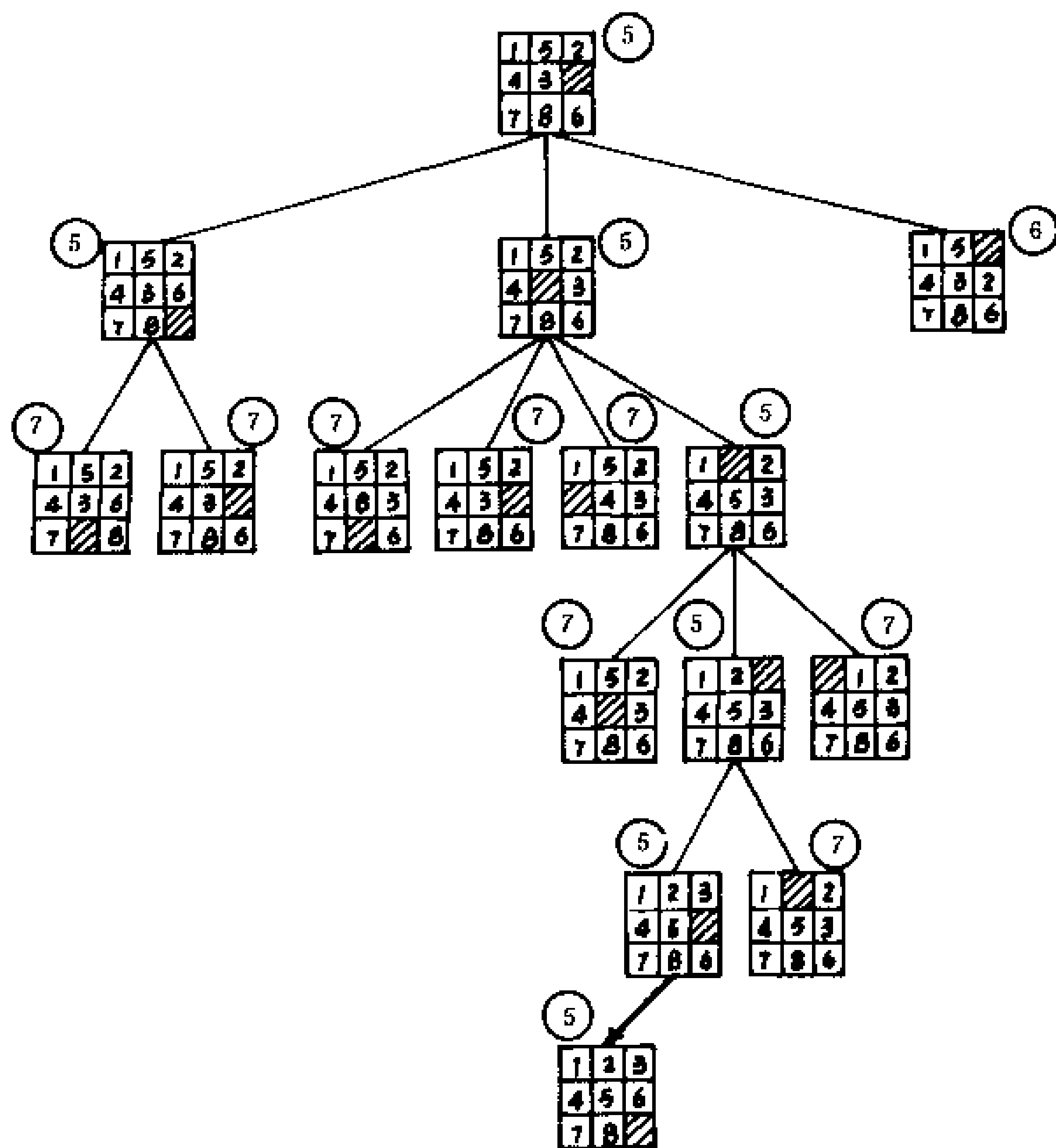


图 13.7 解 8–谜问题的 LC–搜索

13.2.2 分枝限界方法

现在我们正式地定义分枝限界方法。给出一个初始问题和一些想获得最小值的目标函数 f后，分枝限界算法企图直接了当地解决之。如果问题太大，不能马上解决，便将其分解为两个或多个规模较小的子问题，每个子问题包含有一定的约束条件。分解过程重复进行，直到每个未解决的子问题被分解、或者被解决、或者被认为不可能引出原始问题的最优解为止。

①有的初始状态无解，永远达不到目标状态。

在 8-谜例子中，问题是要将牌按行主次序置入棋盘。目标函数 f为将牌排列好所需的移动次数。如果仅有一次移动便可将牌排列好，那么算法立即将问题解决；否则将问题分解生成若干个子问题，对应于每一个合法的移动，生成一个子问题。

正如我们在 8-谜问题已经看到的那样，适用于原始问题的分解过程，可以用一棵有根的树来表示，称作状态空间树。这棵树的结点对应于分解的子问题，而树的边对应于分解过程。原始问题是树的根，树的叶结点是那些可解的或不能进一步分解所抛弃的问题。

应当注意：分枝限界树和分治法表示的树有两点重要的差别，首先，分枝限界树是或树（参见图 13.1），它表示任一个问题的解是原始问题的一个解。因此，不需要检测整棵状态空间树；其次，分枝限界算法表示的树可以是无限的，它表明分枝限界算法没有找到解答。

分枝限界方法的目标是通过检测状态树中的少量子树达到问题求解。设期望的最小代价解是 f^*。对每个分解的子问题，计算下界函数 g，这个下界在给出子问题的约束后，表示该子问题可能的最小代价解。下界函数 $g(x)$具有以下的性质：

(1) g 是一个非降函数。即在状态空间树 T 中，若 y 是 x 的孩子，则 $g(y)\geq g(x)$；

(2) 在每一个表示可行解的叶结点 x 上，$g(x)=f(x)$；

(3) 叶结点表示非可行解时，$g=\infty$。

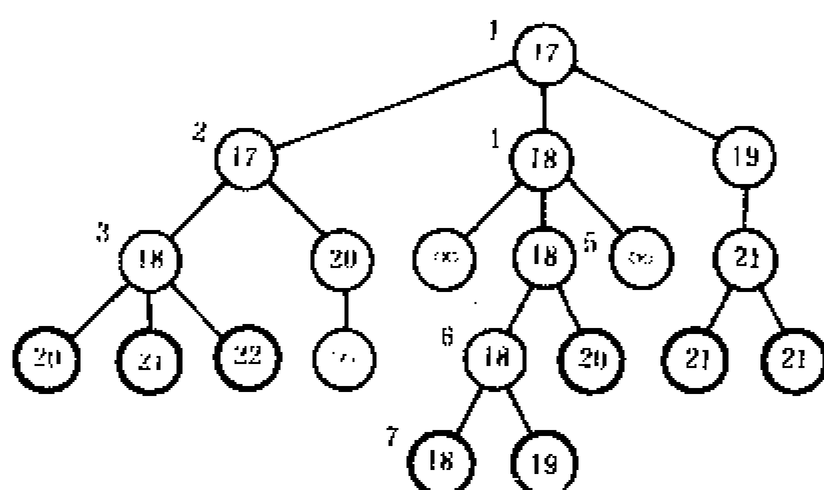


图 13.8 最好优先搜索的状态空间树示例

图 13.8 是状态空间树的另一个例子，结点里面的值是相应子问题的下界。对应于可行解的结点用粗圆圈标明。这个例子代表的问题其代价为 18，即 f^* 的值是 18。

在分枝限界算法的执行过程中，任意一个活结点都存在一个问题的集合，这个集合中的问题已经生成但尚未检测，搜索策略就是确定未检测的子问题的检测次序。最好优先（最好限界）搜索策略使用最小下界来挑选未检测的子问题。图 13.8 中结点近旁的数字标明了最好优先搜索策略检测结点的次序。

下面我们形式化地描述分枝限界算法过程。

算法 13.2 BRANCH-AND-BOUND ALGORITHM

procedure B&B(T,f,g,z);

/ * T 是状态空间树，其根 root 为输入的原始问题；f是输入的目标函数；g 是输入的下界估值函数。z 是输出的最优值。 * /

begin

(1) active__node__set←{root}; / * 初始化 * /

(2) **if** root is a solution node **then** $z\leftarrow f$(root) **else** $z\leftarrow\infty$ **endif**;

/ * 送当前最优解值 * /

(3) **while** active__node__set contains a node x with $g(x)<z$ **do**

$x\leftarrow$node in active node with **min**g( ); / * 选择扩展结点 * /

```
        delete x  from active__node__set;
        for  each child y  of x  do        / * 扩展结点 x * /
          if  y  is a solution & (f (y)<z)   then  z←f (y)   endif;  / * 送当前最优解值 * /
          if y   is not a leaf & (g(y)<z)   then  add y  to active__node__set   endif
        endfor
    endwhile
end .
```

算法开始时，表示原始问题的根是唯一的活结点，把 root 置入活结点集合。扩展结点 x 的意思是：对 x 的每一个孩子；1) 检查 y 是否是一个解结点？若是，则合理地修正当前的最优解值；2) 若下界值小于当前的最优解值（即是 $g(y)<z$）且 y 不是叶结点，则把 y 加到活结点集合中去。因为扩展叶结点是无意义的，所以不能把叶结点加进活结点集合。这样，当前的最优解值越来越小。一旦发现新的解结点 y 时，若它的当前最优解值 $f(y)$小于 z，则立即进行修正。最后得到的 z 值就是最优解值。对每个活结点 y，算法继续执行，直到 $g(y)\geqslant z$ 时算法终止。

总之，分枝限界算法可以表征为：如何生成子问题；如何选择一个特殊的子问题作为继续搜索的出发点；如何抛弃绝望的子问题；如何终止算法的执行。这些步骤中的任何一个都可以并行地实现。

13.2.3 旅行商问题(TSP)

旅行商问题(Traveling Salesperson Problem)简记作 TSP，它是实际应用中出现复杂问题的集中概括和简化形式，是一个比较有趣、容易定义和难于处理的问题，属于 NP−完全问题。

1. 问题描述

设 $G=(V,E)$，$V=\{1,2,\cdots,n\}$是一个有向图，$C=[c_{ij}]$是关于这个图的代价矩阵，其中 c_{ij}是边(i,j)的代价。若边$(i,j)\notin E$，则定义 $c_{ij}=\infty$，并规定所有的 $c_{ii}=\infty$。不失一般性，可以设一切周游路线以顶点 1 为起始点。问题是要从图 G 中找出一条代价最小的周游路线(Tour)。

为了应用分枝限界法，我们必须定义问题的代价函数（目标函数）$f(\cdot)$和下界函数 $g(\cdot)$（“$\cdot$”表示变量为结点），使得对状态空间树上任何结点满足 $g(\cdot)\leqslant f(\cdot)$，其中 $f(\cdot)$的定义和 8−谜问题一样，下界函数我们采用归约矩阵的方法来定义。

2. 归约矩阵[9]

从代价矩阵的任何一行（或列）的各元素中，减去这一行（或列）中的最小元素，称之为对行（或列）归约(Reduction)。第 i 行（或第 j 列）中各元素减去的最小数 $r(i)$（或 $r'(j)$）称作为第 i 行（或第 j列）的约数。如果一个矩阵的各行、各列都是归约了的，那么我们称这个矩阵为归约矩阵(Reduced Matrix)。和数

$$r=\sum_{i=1}^{n}r(i)+\sum_{j=1}^{n}r'(j)$$

称作矩阵 C 的约数。例如，图 13.9(a) 给出了一个有 5 个顶点的图的代价矩阵，图 13.9(b)

是它的归约矩阵，其约数 $r = 25$.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) 代价矩阵

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix} \quad \begin{matrix} r(1)=10 \\ r(2)=2 \\ r(3)=2 \\ r(4)=3 \\ r(5)=4 \end{matrix}$$

$r'(1)=1 \quad r'(3)=3$

$r'(2)=r'(4)=r'(5)=0$

(b) 归约矩阵

图 13.9 代价矩阵 C 及其归约矩阵 C'

对任何一个给定的代价矩阵 C，求它的最小代价周游路线的问题，等价于求它的归约矩阵 C'的最小代价周游路线问题，即有如下定理。

定理 13.1 给定一个图 G 和它的代价矩阵 $C=[c_{ij}]$，设 P 是 G 中任一条周游路线，必有

$$\sum_{(i,j)\in P} c_{ij} = r + \sum_{(i,j)\in P} c'_{ij}$$

这里 $C'=[c'_{ij}]$ 是 C 的归约矩阵，r 是它的约数。

证明 因为任何一条周游路线 P 都必须包含 n 条边，这些边(i,j)所对应的元素 c_{ij}必须是每行和每列有一个且仅有一个①。如果边$(i,j)\in P$，那么 c_{ij} 计入了 P 的代价中，且在第 i 行和第 j 列再不会有其它元素被计入路径 P 的代价中。因为

$$c'_{ij} = c_{ij} - r(i) - r'(j)$$

即

$$c_{ij} = c'_{ij} + r(i) + r'(j)$$

对所有的$(i,j)\in P$，且没有任何行或列会有两个元素对应的两条边包含在 P 中，所以有

$$\sum_{(i,j)\in P} c_{ij} = \sum_{(i,j)\in P} c'_{ij} + r$$

3. 动态状态空间树

假设图 $G=(V,E)$有 m 条边，那么周游路线含有这 m 条边中的 n 条不同的边。我们将旅行商问题的状态空间树动态地构造成一棵二元树，树中结点的左分枝表示周游路线中包含一条指定的边，右分枝表示不包含该条边。这样，在左子树中找周游路线，只需再选取 $n-1$ 条边，而右子树则需对边作 n 次选择。由此可知在左子树中求最优解比在右子树中容易。所以我们希望能选取一条最有可能在最小代价周游路线中的边(i,j)作为这条"分割"边。一般所采用的选择规则是：选取一条使其右

①其逆不一定为真。

子树具有最大 g 值的边。使用这种选择方法可以尽快得到那些 g 值大于最小周游代价的右子树。

对于问题的状态空间树的根结点，显然，其下界 $g(\text{root})=r$ 是正确的，因为任何一条周游路线的代价不可能小于 r。下面，要给状态空间树上的每一个结点建立一个相应的归约代价矩阵。设 A_ω、g_ω 和 r_ω 是状态树上结点 ω 所对应的归约矩阵、下界函数和约数。x 是 ω 的孩子。若将边(i,j)加入周游路线（即 x 是 ω 的左孩子），则计算 x 的归约矩阵 A_x，先从 A_ω 中将第 i 行和第 j列的元素置成∞，且把一切与前面选定的路线构成回路的那些边所对应的元素置成∞。然后，将所得的矩阵进行归约，就是归约矩阵 A_x。约数 r_x 用上面的公式计算。下界函数 $g(x)$的计算公式是：

$$g(x)=g(\omega)+A_x(i,j)+r_x$$

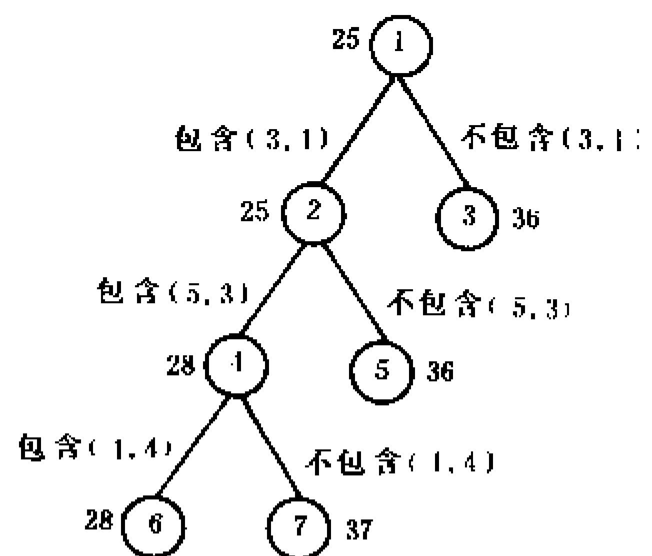


图 13.10 针对图 13.9(a)所示代价矩阵的状态空间树

若 x 是 ω 的右孩子，周游路线中不包含边(i,j)，则将元素 $A_\omega(i,j)$置成∞。重新归约矩阵 A_ω 的第 i 行和第 j列，即是归约矩阵 A_x。计算出 A_x 的约数 r'_x 后，则下界函数计算公式为：

$$g(x)=g(\omega)+r'_x$$

4. 最小代价分枝限界算法的高层描述

算法 13.3 TRAVELING SALESPERSON ALGORITHM (SISD)

输入：代价矩阵 C；

输出：旅行商的最小代价周游路线。

```
begin
    reduce cost matrix, determining the root's lower bound ;
    initially only the root is in the state space tree ;
    /*根表示所有可能的周游路线的集合*/
    while true do
        select unexamined node in the state space tree with the smallest
                    lower bound ;
        if the node represents a tour then exit the loop endif ;
        select the edge whose exclusion increases the lower bound the most;
        for the two cases representing the inclusion and
                        exclusion of the selected edge do
            create a child node with the correct constraint;
            find the lower bound for the child node
        endfor
    endwhile
end.
```

对于图 13.9(a)中的代价矩阵，算法 13.3 所形成的动态状态空间树如图 13.10 所示。根结点的归约代价矩阵见图 13.9(b)，其余各点的归约代价矩阵分别列出如下：

$$\text{结点2}\quad\begin{bmatrix}\infty & 10 & \infty & 0 & 1\\ \infty & \infty & 11 & 2 & 0\\ \infty & \infty & \infty & \infty & \infty\\ \infty & 3 & 12 & \infty & 0\\ \infty & 0 & 0 & 12 & \infty\end{bmatrix}\qquad \text{结点3}\quad\begin{bmatrix}\infty & 10 & 17 & 0 & 1\\ 1 & \infty & 11 & 2 & 0\\ \infty & 3 & \infty & 0 & 2\\ 4 & 3 & 12 & \infty & 0\\ 0 & 0 & 0 & 12 & \infty\end{bmatrix}$$

$$\text{结点4}\quad\begin{bmatrix}\infty & 7 & \infty & 0 & \infty\\ \infty & \infty & \infty & 2 & 0\\ \infty & \infty & \infty & \infty & \infty\\ \infty & 0 & \infty & \infty & 0\\ \infty & \infty & \infty & \infty & \infty\end{bmatrix}\qquad \text{结点5}\quad\begin{bmatrix}\infty & 10 & \infty & 0 & 1\\ \infty & \infty & 0 & 2 & 0\\ \infty & \infty & \infty & \infty & \infty\\ \infty & 3 & 1 & \infty & 0\\ \infty & 0 & \infty & 12 & \infty\end{bmatrix}$$

$$\text{结点6}\quad\begin{bmatrix}\infty & \infty & \infty & \infty & \infty\\ \infty & \infty & \infty & \infty & 0\\ \infty & \infty & \infty & \infty & \infty\\ \infty & 0 & \infty & \infty & \infty\\ \infty & \infty & \infty & \infty & \infty\end{bmatrix}\qquad \text{结点7}\quad\begin{bmatrix}\infty & 0 & \infty & \infty & \infty\\ \infty & \infty & \infty & 0 & 0\\ \infty & \infty & \infty & \infty & \infty\\ \infty & 0 & \infty & \infty & 0\\ \infty & \infty & \infty & \infty & \infty\end{bmatrix}$$

值得指出：图 13.10 的状态树并没画完全。因为从结点 6 往下扩展，只有两层就是最优解答的叶结点，可以对以结点 6 为根的子树中结点逐个检测求解，比计算归约矩阵方法效率高些，对那些矩阵阶数高的问题，在"靠近"解答结点和"没靠近"解答结点分别做不同的处理，效果会更加明显。

5. 两种并行的旅行商问题算法

Mohan[10]设计出了旅行商算法的两种并行化处理。第一种是对 **for** 循环实现并行化，即并行处理某个结点的各个分枝；第二种是并行地执行 **while** 循环，即并行处理多个结点。

前面我们用分枝限界法求解 TSP 时，建立了一棵二元树，该树可用二个处理器来并行处理每一个结点，也就是说，该状态树自然地具有二的并行度。如果同时选择 k 条边来决定其取舍，那么对每个结点将有 2^k 个孩子，需要 2^k 个处理器来并行计算。下面我们给出相应的算法。

算法13.4 TRAVELING SALESPERSON ALGORITHM (TIGHTLY COUPLED MULTIPROCESSOR)

输入：代价矩阵 C；

输出：旅行商的最小代价周游路线。

```
begin
    reduce cost matrix, determining the root's lower bound;
    initially only the root is in the state space tree;
    while true do
        select the unexamined node in the state space tree with the smallest
                lower bound;
        if the node represents a tour then exit the loop endif;
        select the k edges whose exclusion increases the lower bound the most;
        for the 2^k cases representing all inclusion exclusion
                combinations of the selected edges pardo
            create a child node with the correct constraints;
            find the lower bound for the child node
        endfor
    endwhile
end .
```

第二种并行算法产生若干进程，异步地检查子问题所对应的子树，直到找到解答为止。每个进程重复地从未检查的子问题的有序表中移出下界最小者；将问题分解之（除非它能直接求解）；并将新产生的两个子问题插入到有序表中的适当位置上．为了对有序表进行插入和删除，进程控制必须具有排它性，即不允许两个进程同时访问有序表中的某项内容．做这些工作占用的时间相对地小于问题分解所需的时间，因此，有序表的争夺不应该成为速度倍数(加速比)的重要障碍。

图 13.11 列出了在 C_m^* 上的两种并行算法的速度倍数的对比．求解的 TSP 具有 30 个顶点。算法 1 只达到了很低的速度倍数，这是因为增加的处理器花费了大部份时间在生成那些永远不会延伸的结点上（它们的下界太高）．使用 16 个处理器时，算法 2 达到的速度倍数约等于 8。得不到较高速度倍数的障碍主要在于资源共享，即是内部群集器(Intracluster)争夺共享资源，或计算机模块里面的群集器争夺共享资源．这些资源包括 Kmap、Map 总线和目标管理程序。而目标管理程序常常用于产生搜索树的结点。

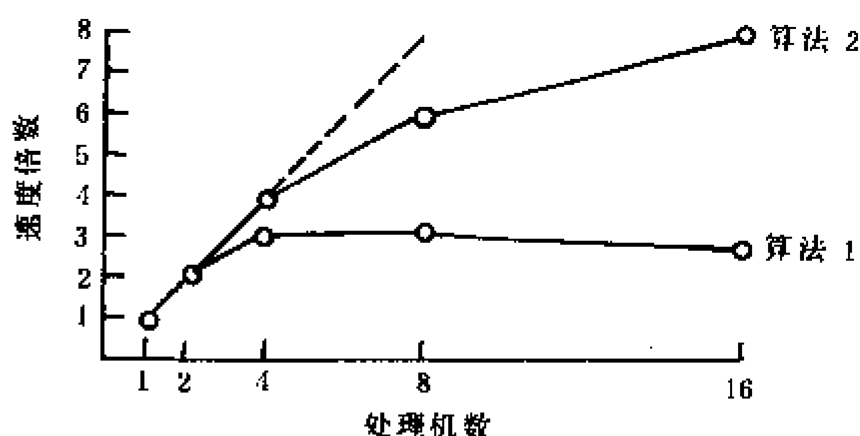


图 13.11　两种并行算法的速度倍数

13.2.4　并行分枝限界算法的异常情况

前面我们介绍了 LC 分枝限界法，叙述了分枝限界算法的并行化，这里着重论述并行最小代价分枝限界执行时可能出现的异常情况和一般情况下的效率问题[11]。

为了论述方便，我们假定最小化目标函数为 $f(x)$．令其最优解值为 f^*．在状态空间树中每一个结点 x，令 $f_{min}(x)=\min\{f(z)$: z 是子树 x 中的可行解$\}$，若不存在这样的 z，

则 $f_{min}=\infty$.

设有 p 个处理器可以利用．令 $q=\min\{p,\ 活结点个数\}$，则 q 个结点选作为下一批的 E–结点（扩展结点）。令 g_{min} 为这 q 个结点中最小的 $g(\ )$值，若这些 E–结点中某个结点为解答结点且其 $g(\ )=g_{min}$，则最小代价解便已找到；否则，这 q 个 E–结点被扩展（一个处理器扩展一个 E–结点），它们的孩子加入到活结点表中。这样的 q 个 E–结点的一次扩展称为最小代价分枝限界并行算法的一次迭代．

对于给定的问题，前面叙述了下界函数满足的性质，在这里再加上一条，加上的这条性质，对于邻近解结点的祖先来说往往是成立的．事实上许多非解答结点可能具有与最优解相同的值．加上这一条乃属于正常现象。总之，下界函数 $g(x)$满足下述 5 条性质：

(1) $g(x)\leqslant f_{min}(x)$：x 为状态空间树中任一结点；

(2) $g(x)=f(x)$：x 为可行解结点（即解答结点）；

(3) $g(x)=\infty$：x 为不可行解结点；

(4) $g(y)\geqslant g(x)$；y 为 x 的孩子；

(5) $g(\ )$值可以相同：即在状态空间树上的若干结点可以有相同的下界函数值。

此外，还可以设置最小代价的上界 u 使算法进一步加速．若 u 是最小代价解的上界，则具有 $g(x)>u$ 的所有活结点 x 可以被抛弃．初始时，u 的值可以置成∞，以后再修改之，比如找到可行解结点 x 时，则令 $u=f(x)$的值，若有更小的可行解值，则再修改 u.

在状态空间树中，结点 x 称为临界的(Critical)结点当且仅当 $g(x)<f^*$。令 $I(p)$表示 p 个处理器可用时所需的迭代次数．则有以下的诸定理．

定理 13.2 给出 $n_1<n_2$ 和 $k>0$，则存在一棵状态空间树使得 $kI(n_1)<I(n_2)$.

证明 考虑图 13.12 所示的状态空间树，所有非叶结点的 g 值等于最小代价解答结点（结点 A）的值 f^*．当使用 n_1 个处理器时，第一次迭代中，某个处理器将根结点扩展，在第二层上产生 n_1+1 个活结点．假定在第二次迭代时，第二层左边的 n_1 个结点被扩展，产生 n_1 个子结点，其中 n_1−1 个因超界而被抛弃，仅剩下一个是活结点．第三次迭代时这个活结点与结点 B（在第二层）一道被扩展．第三层上的这个活结点产生一个解答结点 A，算法终止。故 $I(n_1)=3$.

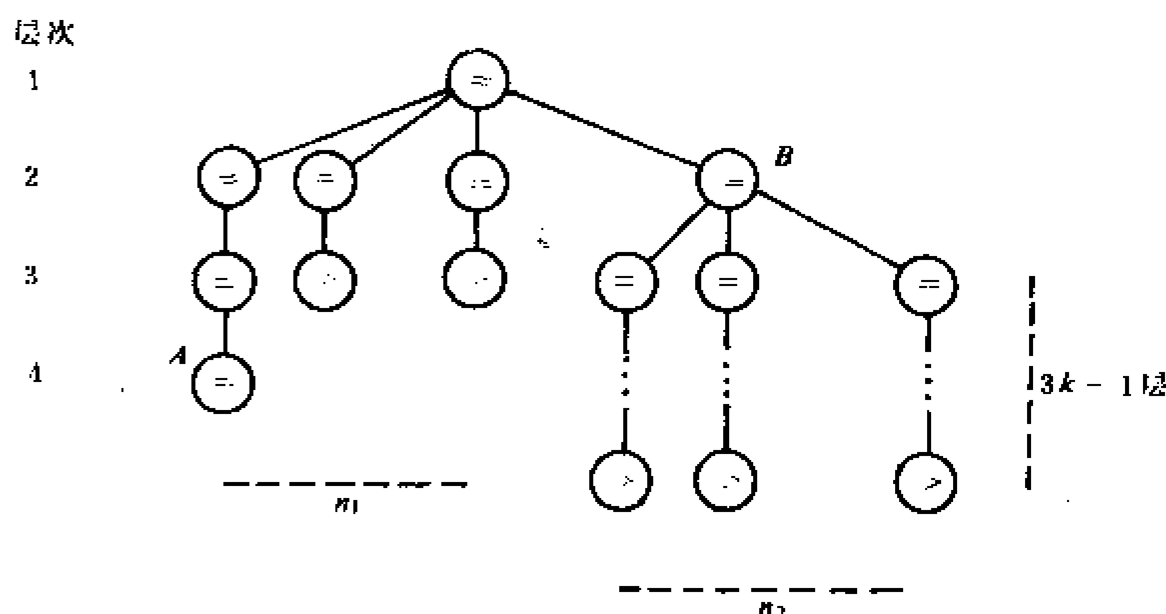


图 13.12 定理 13.2 的实例

如果使用 n_2 个处理器，第一次迭代情况与上面一样，产生 n_1+1 个子结点，这 n_1+1

个子结点在第二次迭代时全部被扩展．于是在第三层上产生 n_2+1 个活结点，共有 n_1+n_2 个活结点．显然，第三次迭代只能扩展 n_2 个活结点，完全可能是右边的 n_2 个结点被扩展．接下去继续进行迭代，第 4，5，…，$3k$ 次迭代完全限制在根结点的最右子树上，最后在第$(3k+1)$次迭代时，最小代价解结点 A 才能创建．因此 $I(n_2)=3k+1$．

对比两种结果，我们得到 $kI(n_1)<I(n_2)$。

这个定理告诉我们，执行并行最小代价分枝限算法，增加处理器不一定都是有效的．这是因为增加处理器后，可能导致貌似理想的结点的生成，从而导致相当数量或所有处理器去搜索其子树却又一无所得．当处理器个数较少时，遇到貌似理想的结点 x 被扩展时，上界 u 已经修改且 $u\leqslant g(x)$，所以会抛弃结点 x，执行时间反而少．

当存在大量结点的 g 值等于 f^* 时，定理 13.2 指出可能出现异常情况。事实上，即使只有一个处理器处理这样的结点，也可能会引起麻烦。如果我们要求只有最小代价解答结点才能有等于 f^* 的 g 值，那么定理 13.2 不再成立。特别地，若 $n_1=1$，则处理器数 n_2 的增加不会引起迭代次数的增加，这就是下一条定理要证明的结论。

定理 13.3 如果 x 不是解答结点时有 $g(x)\neq f^*$，那么对所有 $n>1$ 有 $I(1)\geqslant I(n)$。

证明 当使用一个处理器时，只有临界结点和最小代价解答结点才能成为 E–结点，这是因为任何时候选择一个 E–结点时，活结点表中肯定至少有一个结点 x 满足 $g(x)\leqslant f^*$．而且在算法终止之前，任何临界结点将成为 E–结点．所以，若临界结点共有 m 个，则 $I(1)=m$．

当使用 n 个处理器时$(n>1)$，一些非临界结点将成为 E–结点。但每次迭代中至少有一个 E–结点是临界结点，故 $I(n)\leqslant m$．因此 $I(1)\leqslant I(n)$．

下面的定理证明：为了找一个解答结点，增加处理器的数目，实际上可能导致迭代次数的不成比例的减少。

定理 13.4 若给出 $n_1<n_2$ 和 $k>n_2/n_1$，则存在一棵状态空间树使得 $I(n_1)/I(n_2)\geqslant k>n_2/n_1$。

证明 参见 Lai 和 Sahni 的论文[11]．

定理 13.5 如果 x 不是最小代价答案结点时有 $g(x)\neq f^*$，那么对 $n>1$ 有 $I(1)/I(n)\leqslant n$。

证明 由定理 13.3 的证明可知：$I(1)=m$，其中 m 是临界结点数；因为所有临界结点在算法终止时必须变成为 E–结点，所以 $I(n)\geqslant m/n$．因此 $I(1)/I(n)\leqslant n$．

Lai 和 Sahni 在某些 0／1 背包问题的实例中已发现异常性质，但是他们得到的结论是：异常性质实践中很少出现，而且通常是：(1) 假定问题足够大时，增加处理器数目不会增加执行时间；(2) 不可能指望超线性的速度倍数．

13.3 α–β 搜索

国际象棋、围棋、五子棋和中国象棋都属于二人零和博弈，这种博弈由两个选手对弈，两人依次走子，一个选手的损失是另一个选手的受益，反之亦然．最后结果是一位选手赢、另一位选手输，或者双方认和．博弈是一种竞争，竞争现象广泛存在于社会活动的许多方面，例如，政治、经济、军事、外交、科技等领域都存在着竞争现象．

在计算机上玩博弈要在博弈树上进行搜索，以确定下一步如何行动最有利。博弈树是一棵与／或树，树中结点代表棋盘位置(Position)，根结点代表当前的棋盘位置，边代表移动(Move)棋子。除了根结点，其余结点都是在搜索过程中动态地生成和删除。博弈树的搜索是运用修剪来减少计算量。它的修剪方式称为 $\alpha-\beta$ **修剪**($\alpha-\beta$ Pruning)。$\alpha-\beta$ 搜索算法运用上、下界和自底向上计算得到的求值函数（启发函数）值比较，去掉超出界限的结点，即是说，该算法一旦确定某子树与搜索结果无关时，就将该子树剪去。这样可以加快搜索速度。

下面，我们先介绍串行的 $\alpha-\beta$ 算法及其改进技术，这些改进技术的目的相同，都是希望尽早更多地剪掉与搜索结果无关的子树。然后介绍并行的 $\alpha-\beta$ 算法，并行算法采用了某些改进技术。一般地说，k 个处理器并行执行算法并不能得到 k 倍的速度倍数，这是因为并行机制引入了其它额外开销，其中最主要的是**通信开销**(Communication Overhead)和**搜索开销**(Search Overhead)。在搜索过程中，各处理器之间要交换信息，这方面的开销叫做通信开销。由于使用并行技术，在并行搜索时会出现冗余搜索，即一些在串行搜索时被剪去的子树在并行搜索中会进行搜索。这方面的开销叫做搜索开销。通常，通信开销和搜索开销成反比关系，减少通信开销就增加搜索开销，减少搜索开销就增加通信开销。各种并行算法按照合理的硬、软件配置，折衷综合两种开销的影响，得到不同的搜索性能指标，取得了满意的结果。

13.3.1 $\alpha-\beta$ 算法

通常对博弈树搜索，弈者搜索的深度越深，博弈的质量就越高①。$\alpha-\beta$ 算法力图避免搜索那些与结果无关的子树，导致搜索深度的加深。下面描述的 $\alpha-\beta$ 算法[6]包含四个变参数：p—当前的位置；*alpha* 和 *beta*—搜索取值的范围；*depth*—搜索的深度。算法用函数形式给出，它返回位置 p 的最小最大值。算法中调用四个函数：即求值函数 EVALUATE，生成结点函数 GENERATE，做移动函数 MAKE 和不做移动函数 UNDO。这个算法的优点是：毋需明显的测试谁在进行移动，都能使其达到最小值和最大值。

算法 13.5 ALPHA-BETA ALGORITHM(SISD)

```
function ALPHA__BETA(p, alpha, beta, depth);
begin
  if depth≤0
    then return (EVALUATE(p))   /＊求终端结点的值＊/
    else
      width←GENERATE(p));
        /＊生成 p 的孩子 p1, …, pwidth, width 是合法的移动数＊/
      if width=0
```

①由于求值函数不是绝对精确，亦存在反常现象。Nau[17]通过理论分析指出，在下棋程序中，增加搜索深度不一定增加取胜的可能性。

```
        then  return(EVALUATE(p))   /* 没有合法的移动 */
        else
         score←alpha;
         for i←1 to width do
           MAKE(p_i);  /* 做移动 */
           value←-ALPHA_BETA(p_i,-beta,-score,depth-1);
           UNDO(p_i); /* 不做移动 */
           if value>score then score←value endif;  /* 已找到较好的移动 */
           if score>beta then  return(score) endif  /* 已经找到剪枝 */
         endfor
        endif
    endif;
    return(score)
end.
```

为了说明算法 13.5 的执行过程，考虑图 13.13 所示的博弈树。树中每条边上的一对数分别表示双亲结点传送给它的孩子结点的 alpha 和 beta 值。但传送给叶结点的值没有标明，这是因为它们是不相关的。若 $depth \leqslant 0$，则求值函数往往确定了位置的值，且这个值立即返回给双亲结点。结点中的数字是求值函数的值，它们是按照先移动的弈者画出的。从根结点开始，奇数层为先移动弈者的值，偶数层为对手的值。

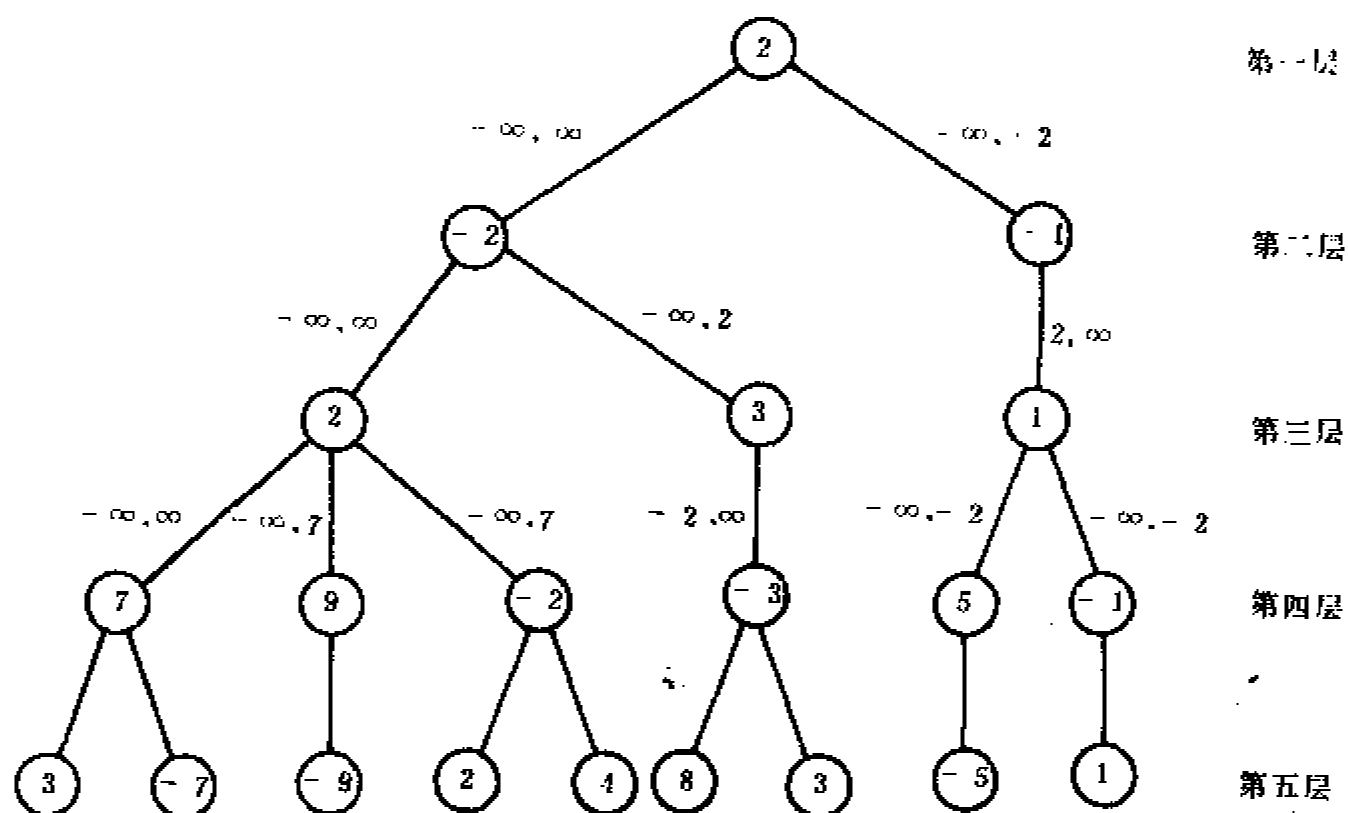


图 13.13 α-β 搜索的例子

每一个内部结点表示博弈中的一个位置。当搜索到达某个内部结点时，由于前面已经考虑过的移动选择至少使弈者领先 alpha 值，而对手决不允许从当前位置让弈者获得大于 beta 值，因此 alpha 和 beta 决定了搜索的一个窗口。若从这个位置出发不能导致值大于 alpha 的移动，则弈者将不跟着进行活动。若任何移动导致其值大于 beta，则弈者的对手

将确信决不能到达这个位置。于是算法 13.5 找出孩子结点返回的负值中的最大值。但是，若任何返回值的负数大于 beta，则搜索停止且修剪剩余部分。

13.3.2 改进的 $\alpha-\beta$ 搜索

这里我们简单地叙述改进算法的基本思想，详细算法参见文献[12]。

1. 吸出搜索(Aspiration Search)

如上所述，$\alpha-\beta$ 算法定义了一个窗口，记作 (α, β)。窗口越小，搜索时可能被剪掉的子树越多，搜索速度越快。在实际应用中，为了得到正确的根值，初始窗口可选用(−INF,INF)，这里 INF 是求值函数可能返回的最大值.

吸出算法的思想是使用小窗口来加速搜索．它的主要步骤如下:

首先用某种方法估计根值 U 及其误差 ε，以$(U-\varepsilon, U+\varepsilon)$为初始窗口，调用 $\alpha-\beta$ 算法搜索博弈树．若返回值低于窗口的下限，则用$(-\mathrm{INF}, U-\varepsilon)$作为窗口重新进行 $\alpha-\beta$ 搜索；若返回值高于窗口的上限，则用$(U+\varepsilon, \mathrm{INF})$作为窗口重新搜索，如此继续进行，最后总能得到正确的根值.

倘若缩小窗口节省的开销大于重新搜索增加的开销，则搜索速度可以加快。实际上，可使得$(U-\varepsilon, U+\varepsilon)$比$(-\mathrm{INF}, \mathrm{INF})$小得多．用可靠方法估计根值 U 和误差 ε 组成的窗口$(U-\varepsilon, U+\varepsilon)$，包含真正的根值的概率也较大，吸出搜索是比较有效的.

2. 转换表(Transposition Table)

转换表是一个大的散列表，表中存贮那些已被搜索过的结点和相应的一些信息．在博弈树的搜索过程中，当搜索到某个结点时，首先在转换表中检索该结点。若发现该结点已经搜索过，则利用表中存贮的一些信息来加速对以该结点为根的子树的搜索，有时甚至可以删除该子树，用表中存贮的以前搜索的结果作为本次搜索的结果．由于在搜索一棵博弈树时，已被搜索过的结点再次出现是常有的事，特别是使用杀手启发(Killer Heuristic)、迭代深化技术时，这种情况更是常见的．因此，转换表是较有用的.

使用转换表开销小，且潜在的收益大，但存在着存贮管理问题．另外，转换表可以使得博弈树更加有序．这是因为若发现结点已被搜索过，即在转换表中检索到该结点，则可以先检查表中记录的移动，而这个移动很可能是最好的移动，因为它是前面的搜索找到的最好移动.

3. 迭代深化(Iterative Deepening)

迭代深化是指进行深度为 D 层的搜索之前先进行深度为 $D-1$ 层的搜索，$D=2,3,\cdots,\mathrm{maxdepth}$，其中 maxdepth 是最大搜索深度。除第一次搜索外，每次搜索都要利用前一次搜索的结果．利用时采取的方法较多．例如，$D-1$ 层深的搜索结束后，保留主要变量(Principal Variation)，并把它作为 D 层深搜索的初始序列，由于 $D-1$ 层的主要变量很可能是 D 层主要变量的一个组成部分，因而 D 层搜索可以快速地进行、又如，可以把 $D-1$ 层搜索得到的根值作为对 D 层深的搜索根值的估计值，以它为中心设置一个狭窄的窗口，用吸出算法进行 D 层深的搜索，同样也可以加快搜索速度.

迭代深化技术的主要优点有两个：其一是便于时间控制．在博弈竞赛中，每一步棋允

许用的时间有限制，因而搜索时间也有限制．而且不同的博弈树的搜索时间相差悬殊．因此，搜索深度较难掌握．若取得太小，则浪费许多时间；若取大了，则搜索某些树时会出现时间限制到而得不出可靠的搜索结果的现象．使用迭代深化搜索时，如果时间限制到时正在进行 D 层深的搜索，那么至少可以保证 $D-1$ 层的主要变量是可靠的．

其二是迭代深化可以使转换表或杀手表充满有用的移动和相应的信息。若迭代深化和转换表或杀手启发结合作用，则可更加提高搜索的速度．且每次迭代后对移动进行排序，可使得博弈树更加有序．

4. 主要变量搜索(Principal Variation Search)

α-β 算法的一个有趣的实现是用一种特别的方法处理第一个变量．主要变量搜索（简记作 PVS）正是这样做的．

PVS 的基本思想是：假定第一次做的移动在每个结点上是好的，于是 PVS 对第一棵子树作递归的调用，并确定它们的值，其余的子树依次进行检查．若这些子树中的某一个有值大于当前的主要变量，则将它作为新的主要变量，并用正确的窗口再进行搜索．换句话说，一旦得出主要变量，就用最小窗口(−score−1，−score)搜索树的剩余部分，这里 score 是当前找到的最好值．如果发现子树中某一棵的次序比它的兄弟次序好，那么必须再进行搜索．PVS 对好的有序（非随机）博弈树搜索证明是有效的．当迭代深化常用于提供一个主要变量时，PVS 变得更加有效．

由此可见，PVS 方法的实质是：假定有序的博弈树中检查最左路径是最优的，并利用这条路径上的值形成狭窄的窗口，搜索剩余的子树，以试图尽快地证明它们是低级的。若一棵子树被证明是高级的，则必须重新搜索以确定它的真正值．在这种情况下，以前的许多搜索也许要报废．因此，PVS 检查的结点数也可能比 α-β 搜索还多，这是 PVS 的一点风险．但在好的有序树上搜索时，由于确定子树是低级所节省的搜索时间会超过识别高级子树后做无用搜索的代价，PVS 是比较有效的．

13.3.3 并行搜索算法

并行搜索算法通常分为三类：**窗口分裂**(Window Splitting)或**并行吸出**(Parallel Aspiration)算法，**并行求值**(Parallel Evaluation)或**并行计算结点函数**(Parallel Calculation of Node Function)和**树分解**(Tree Decomposition)算法．

窗口分裂是将初始窗口(−INF，INF)划分成互不相交的区间分给各个处理器，每个处理器以分给它的区间为初始窗口用 α-β 算法搜索同一棵博弈树．由于这些区间的并集复盖着初始窗口(−INF，INF)，最后总有一个处理器搜索到真正值。例如，设有 5 个处理器，可以分别以区间(−INF，$U-3\varepsilon$)，($U-3\varepsilon$，$U-\varepsilon$)，($U-\varepsilon$，$U+\varepsilon$)，($U+\varepsilon$，$U+3\varepsilon$)，($U+3\varepsilon$，INF)，为初始窗口去搜索博弈树．使用这种并行搜索方法，当处理器数 k 较小时，比如 $k=2$ 或 3，可使速度倍数超过 k．但是，这种方法所能达到的最大速度倍数局限 5～6，并和使用多少处理器无关[13]．

对终端结点的求值往往是很费时间的，这是因为要使求值准确，往往是计算求值函数的工作量很大．并行求值算法的基本思想是：把求值函数设计复杂些，分成许多部份交给不同的处理器并行计算，最后把计算结果综合起来，这样可提高搜索性能；或者，并行计

算各个结点的求值函数，自底向上的比较后得出根结点的值，这样可以使搜索加深。

树分解是指把不同的子树交给不同的处理器并行搜索。这种方法引入大量冗余开销，主要是搜索开销和通信开销。实现树分解算法通常要使用处理器树（即处理器按树状连接）。下面我们介绍树分解算法的几种实现。

1. 树分裂算法(Tree–Splitting Algorithm)[12,4]

这个算法是在处理器树结构上实现的。为了限制处理器间通信，应当使用简单的连接结构，比如二叉树结构，结点代表处理器，边代表处理器间的连接。在最简单的情形，所有非终端结点处理器执行主算法，一旦从双亲处理器处接收一个位置和(α,β)窗口，则生成后继者位置，并分派它们给孩子处理器去并行搜索。无论什么时候一个孩子完成，就返回一个值作为它的子树的值。若这个值导致α界限的变化，则双亲处理器中断其它孩子的搜索，并强迫孩子修改α–β值。修改算法如下：

算法 13.6　UPDATE $\alpha-\beta$ WINDOW ALGORITHM

```
var alpha, beta:  array[1..maxdepth] of integer;
/ * alpha–beta bounds are stored in global tables * /
procedure UPDATE (depth, side, bound: integer);
begin
  if (side > 0) then alpha[depth]←max(apha[depth].bound)
               else beta[depth]←min(beta[depth].bound)
  endif;
  if (depth > 0) then UPDATE(depth−1,−side,−bound) endif
end.
```

终端结点的处理器亦接收一个位置和窗口，但是简单地执行$\alpha-\beta$算法，使得构造的博弈树达到它的最大允许深度。它计算树的终端结点值，并返回最好值给双亲结点处理器作为该子树的值。

树分裂算法形式地描述如下：

算法 13.7　TREE–SPLITTING ALGORITHM

```
function TREESPLIT(p:position; alpha, beta: integer): integer;
  var width, i: integer;
      value: array[1..maxwidth] of integer;
      j : processor;
  begin
    if (i is a leaf processor) then return (ALPHA_BETA(p,alpha, beta)) endif;
    width←GENERATE(p);  / * 生成后继者 p[1]···p[width] * /
    for i←1 to width pardo
      while (a child processor j is idle) do
        value[i]←–j· TREESPLIT(p[i],−beta,−alpha);
        begin     / * 进入临界区 * /
          critical
          if (value[i] > alpha) then alpha←value[i] endif
```

```
        end;
        if (alpha ≥ beta  then terminate( ); return (alpha)  endif    / * 要剪枝吗? * /
      endwhile
    endfor;
    return (alpha)
  end.
```

算法 13.7 的几点说明如下:

(1) j• TREESPLIT 表示在第 j 个处理器上递归执行树分裂算法;

(2) **for–pardo** 并行执行迭代中的每个进程，直到全部迭代完成时只有一个进程为止;

(3) **while–do** 表示处理器 j 空闲时则执行循环体，若全部处理器不是空闲的，则处于等待状态;

(4) **critical** 表示一个时刻仅允许一个进程进入到临界区域;

(5) terminate 表示中断在 **for** 循环中仍处于活动状态的处理器搜索.

并行实现的一个重要特性是动态的修改 $\alpha-\beta$ 窗口，这是它加快孩子处理器完成的原因。即使无浪费的结构，动态地共享这些边界亦是可取的。但是，树分裂算法实现中，仍然花费了大量时间做无用功。这是因为此算法的搜索策略比较死板，等同地对待搜索树中的各个结点，在最好或较好的移动找到前，大量的子树已经检查过，而在串行实现时，许多子树是可以修剪的。因此，搜索开销被大量引入，成为效率损失的主要来源。幸运的是：算法 13.6 所示的修改方法相当简单。

P_1 P_2 P_3 P_{n-1} P_n

图 13.14　进行第 n+1 次迭代搜索时的部分博弈树算法

2. 主要变量分裂(Principal Variation Splitting)

主要变量分裂简记作 PV–分裂，亦使用处理器树来实现，假定处理器树的深度小于博弈树的深度。并假定搜索博弈树中的一个结点的所有移动，是按从左到右的顺序排列，即最左移动是第一个移动，最右移动是最后一个移动，排列的顺序按照这些移动的求值大小，从大到小排序。

PV–分裂算法的实质在于：假定最左路径是最好的移动序列，首先检查这条路径，并用检查的结果构成较小的窗口去检查其它的移动。如图 13.14 所示，所有处理器沿最左路径 $P_1P_2\cdots P_n$ 向下检查，一直到达 P_n，把 P_n 的孩子分给 n 个处理器并行搜索，搜索结果返回到 P_n；用 P_n 搜索结果构成的小窗口交给各个处理器，将除 P_n 外的所有 P_{n-1} 的孩子分给这些处理器并行搜索，结果返回给 P_{n-1}，…，一直到 P_1 的孩子都被检查并返回结果给 P_1 时，搜索则完成。显然，这个搜索算法对强有序树将是有效的。为了得到强有序，可以利用迭代深化技术。

PV–分裂的并行实现形式化描述如下:

算法 13.8 PARALLEL PV-SPLITTING ALGORITHM

```
function PV__SPLIT(p: position; alpha, beta, length: integer) : integer;
  var width, i: integer;
    value: array [1..maxwidth] of integer;
    j : processor;
  begin
    if (length ≤ 0) then   return (TREESPLIT(p,alpha,beta)) endif;
                           / * 处理器树的终端结点 * /
    width←GENERATE(p);     / * 生成一个指针指向后继数组 * /
    alpha←—PV—SPLIT(p1, —beta,—alpha, length—1);
    if (alpha ≥ beta) then return(alpha)  endif;
    for i←2 to width pardo      / * 并行找后继 * /
      while (a slave j is idle)  do / * 有空闲处理器时做下面语句组，否则，等待 * /
        value[i]←—j .TREESPLIT(p[i],—beta, —alpha);
        begin critical
           if  (value[i] > alpha) then alpha←value[i] endif
        end;
        if (alpha ≥ beta) then  terminate( ); return(alpha) endif
      endwhile
    endfor;
    return(alpha)
  end.
```

算法的说明和算法 13.7 一样。[15]中给出了此算法的理论分析模型以及实际搏弈程序中应用的测试结果。测试结果显示，PV—分裂算法搜索开销很小，如图 13.15 所示，这是因为它沿主要变量搜索和在测试实例中使用了转换表、排斥表技术。另方面通信开销却是效率损失的主要因素，这里的通信开销包括检索、修改存贮表格（指转换表和排斥表）和传送信息的时间以及同步开销。所谓同步开销是指为了等待别的处理器完成搜索而空闲的时间开销。在 PV—分裂算法中，同步开销又占主要地位，因为它要求同层之间、同次迭代之间进行完整的信息传递。各处理器此时必须保持同步，所以导致了大量的同步开销。从图 13.15 可以看出：随着处理器数目的增加，搜索开销平缓地增长，而通信开销却近似地成指数增长。

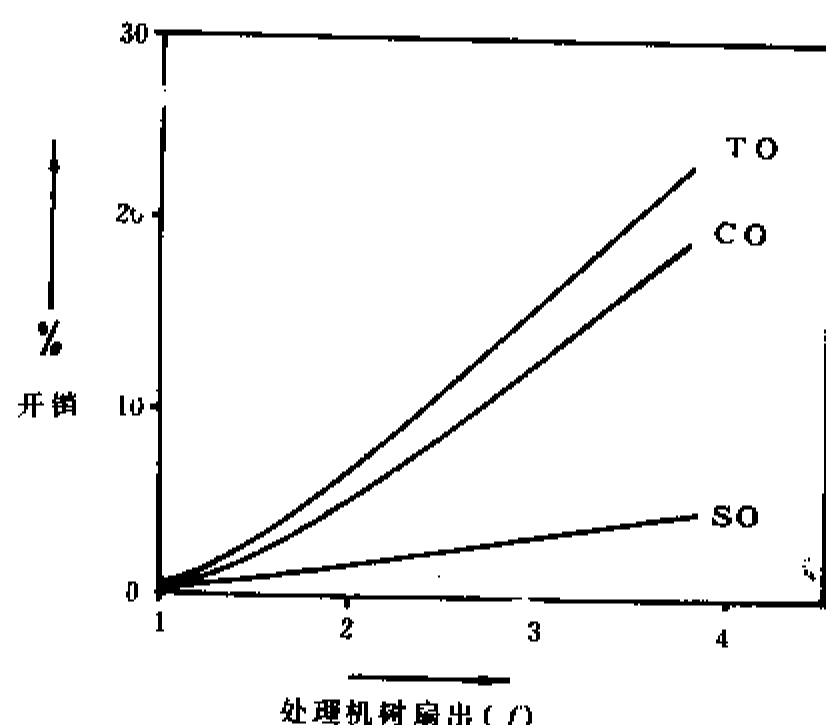


图 13.15　5 个弈者搜索的平均开销(图中：TO 为总开销，SO 为搜索开销，CO=TO—SO)

3. 异步迭代深化

异步迭代深化并行 $\alpha-\beta$ 搜索(Unsynchronized Iteratively Deepning Parallel Alpha-Beta Search，简记为 UIDPABS)是一个比较简单的树分解算法。搜索时不需要通信，也不需要保持各处理器间的同步，因而控制简单。下面简单地描述该算法的工作过程，详细的结果参见文献[16]。

算法 13.9 UIDPABS ALGORITHM

(1) 在头两次迭代中，所有处理器执行相同的搜索，把根结点处的移动排次序，为后面的搜索做好准备。因为全部处理器的排序标准都相同，所以它们最后得到相同的移动序列。

(2) 确定搜索窗口：每个窗口对应一个变量 score，窗口力求用窄小的。窗口宽度的大小需参考以前移动的 score 计算值和第二次迭代找到的 score 值，若两者之差小于一个孩子处理器的计算值，则置窗口为$(SR-P+1, SR+P-1)$；否则置为$(SR-2P+1, SR+2P-1)$。其中：SR 为根处的 score 值，P 为分配给孩子处理器计算的 score 值。

(3) 平均分配各处理器在根处的移动，若有 K 个处理器，根的移动有 M 个，则第 k 个处理器的移动为 k，$k+K$，$k+2K$，…，$k+\lfloor (M-k)/K \rfloor \cdot K$。这样，前两次迭代中所发现的具有高值的 score 移动被均匀地分配给各个处理器。

(4) 各处理器分别对所分配的移动做迭代深化搜索。即迭代深化搜索在各处理器上是异步进行的。

(5) 若迭代结束时处理器计算的 score 值小于窗口对应的 score 值，则下一次迭代用较小的值代替窗口的 score 值，且窗口宽度不改变。然而，若以后的迭代中又出现 score 值小于窗口的，则不再改变窗口的 score 值；若在迭代中间出现处理器计算的 score 值大于窗口的值，则迭代立即停止，并改变窗口重新开始，比如若用窗口(x, y)使迭代中途停止，则用窗口(y, INF)重新开始迭代。因此，在搜索过程中，处理器并不是都在搜索同一次迭代的移动，或者利用相同的窗口搜索。

(6) 当到达某个事先确定的时间限制 T_{max} 时，则搜索终止。每个处理器返回它的主要变量和 score 值给主处理器，主机从中确定最好的并将其决定通知其它处理器。若一个处理器正在搜索先前的移动子集而没有本次迭代的主要变量和 score 值，同时收到主机的搜索终止命令时，则将倒数第二次迭代中找到的主要变量和 score 传送给主机。如果主机自已找到最好的移动，即主机有一个 score 值在上、下界内，那么搜索完成，此时主机将计算的 score 值记人窗口和发送给别的处理器。如果全部处理器计算的 score 值都小于各自窗口的对应值，那么整个搜索必须用更宽的窗口重新进行。

UIDPABS 的突出特点是异步进行搜索，而且算法控制简单。但是冗余搜索增多，搜索开销增大，因而导致性能下降，和 PV-分裂算法相比，在 8 个处理器的情形，PV-分裂比 UIDPABS 更为有效，大约是高 30%。

13.4 小 结

本章主要讨论用状态空间树表示的组合搜索问题的并行算法。对“与树”的搜索，我们

介绍了基于分治策略设计的并行算法．由于传播开销和组合开销，并行分治算法的速度倍数受到了限制。并行计算的效率和并行的粒度有很大关系。

对于"或树"的搜索，这里叙述了分枝限界方法．在最小代价（或叫最好优先）分枝限界算法中，每一步选择最小代价的结点进行扩展，求解过程中搜索的结点数目最少．表征分枝限界算法的四个要素（即分枝规则、选择规则、消去规则和终止条件）都可以并行实现，对求解旅行商问题，这里介绍了 Mohan 的两种并行化处理，即使用要素中的前两条规则，并行分解问题为更小的子问题和并行安排搜索次序．并行分枝限界算法的主要问题是如何保持处理器的高效率，合理地结合并行和选择结点的执行次序，并行算法可望达到好的效率。然而，Lai 和 Sahni[11]已经证明：并行分枝限界算法存在着异常现象，且在解 0／1 背包问题中业已出现。尽管如此，分枝限界的并行化仍然经常被使用，而且在大多数实践中很少出现异常情况．

博弈树是"与／或树"的很好例子。对与／或树的搜索经常使用 $\alpha-\beta$ 搜索，它运用上、下界和自底向上计算得出的启发函数值比较，剪去超出界限的结点。它本身就是一个有效的方法．本章 13.3 节除了介绍 Knuth 和 Moore 的 $\alpha-\beta$ 搜索算法外，还叙述了吸出搜索、转换表、迭代深化、主要变量搜索等改进技术，这些改进方法是并行化可依照的．关于 $\alpha-\beta$ 搜索的并行化，这里着重介绍了树分解算法，包括树分裂算法、PV−分裂算法和异步迭代深化算法 UIDPABS．应当指出：在多处理机上的并行 $\alpha-\beta$ 搜索，增加了额外的开销，即是搜索开销和通信开销．树分裂算法和 UIDPABS 算法的搜索开销是主要的损失效率的来源，PV−分裂算法主要是通信开销影响效率的提高．

关于组合搜索问题，我国计算机科学家李国杰研究员进行了系统全面的研究，他和他的同事发表了一系列的有关论文，在并行处理领域影响较大，读者若能阅读他们的论文，对组合搜索问题的并行算法，将会有更全面的理解．

基于分治策略设计的并行算法，本书前而的章节已经涉及．Horowitz and Zorat[3]讨论了矩阵乘法和排序算法，在他们建议的模型上达到了最优的复杂性．Das and Deo[19]在 SIMD−EREW PARM 模型上，应用分治策略建议了求无向图连通分支和最小生成森林的并行算法，又使用它们设计了求图的基本回路集、桥以及检测二分图的并行算法，无论是对稠密图还是对稀疏图，所有这些算法都是最优的．

关于最小代价分枝限界并行算法的效率分析．对于共享存储模型，李国杰及其同事推导的性能界限为[1,22]：

$$\frac{T_b(1)-1}{k}+1 \leqslant T_b(k) \leqslant \frac{T_b(1)}{k}+\frac{k-1}{k}h$$

其中：

$T_1(b)$ 是单个处理器执行搜索所需的时间；

$T_k(b)$ 是 k 个处理器执行搜索所需的时间；

k 是处理器个数；

h 是状态空间树的高度。

并且指出：当下界函数有中等精度时，最小代价搜索得出好的性能．Quinn[23]在超立方体结构计算机上实现了并行分枝限界算法．Wah 等人在一种特别设计的计算机上实现了最

好优先的分枝限界算法，并描述了动态规划的并行化和并行解逻辑程序设计问题．Yu 等人[24]使用好的存贮管理系统，以便支持大型存贮空间的需要，在二级存贮系统上有效地实现了并行分枝限界算法．

并行搜索与／或树的文献较多，不同的并行算法为了达到各自的目标，采用不同的策略得出相应的结果．但是，并行 $\alpha-\beta$ 搜索算法的速度倍数，在实践中究竟能达到多少，仍然是一个尚需研究的问题，对博弈树的并行搜索，Marsland & Campbell[12]已发表了可读性好的综述论文，建议读者阅读．

关于某些与／或树的启发性搜索，李国杰等人对组合判定问题的研究后提出：决定下一步做什么取决于两方面因素，即每个分枝的成功概率 P 及搜索每一子树的成本 C（时间），对于一组具有"或关系"的结点应选用启发函数 P/C 最大的结点先做；对于一组具有"与关系"的结点应选择$(1-P)/C$ 最大的结点先做．文中还建议了一种最优的搜索方案，将"解答树"按成功率排序，依次地搜索各棵解答树．然而，在实际搜索中，面对大量的结点存在面每个结点的成功概率都较小时，如何正确使用"或并行"和"与并行"，仍然是一个悬面未决的问题．

许多人工智能问题都可归结为在一个巨大的可能解的搜索空间中寻找一个或若干个满意解．探试搜索①与并行处理技术是提高智能机效率的基本途径．由于探试搜索算法在许多方面不同于通常的确定型算法，将并行技术与探试搜索结合起来必然遇到许多新问题．对这些问题的深人研究将为研制智能计算机提供必要的理论基础，减少设计的盲目性．下面我们扼要地列出几个有关并行组合搜索的重要研究课题[25]．

1. 如何表示与评价启发知识

指导搜索的启发知识实际上是一种控制元知识，即关于如何运用问题领域知识的知识和关于问题求解方法的知识．在计算机内如何简洁有效地表示这些知识是至今尚未解决的问题．搜索效率在很大程度上决定于问题表示的方法．问题表示方法又与知识表达方式有关，所以研究搜索必须与研究知识表达相结合．在串行条件下如何比较两个启发函数也有一些成果，但这些结论一般不能直接用于并行搜索，我们需要建立一套在并行环境下如何评价启发函数的理论．

2. 通过机器自学习积累完善启发知识

启发知识是不精确和不完全的．积累与完善启发知识的重要途径是机器自学习．而机器学习本身也常常采用归纳搜索的办法，从事例中归纳出有用的启发知识．这些启发知识不仅可用于指导搜索路径的选择，而且可用于建立优势关系，提高淘汰率。一个智能应用程序一开始可采用盲目搜索，通过运行逐渐积累启发信息和知识，提高智能程度．事实上，目前研制的神经网计算机都包含了机器学习功能。有些是学习输人与输出之间的关联性；有的学习识别输人信息中有意义的模式．这些已采用的学习机制的速度一般都很慢，而且学习的速度取决于不能自动改变的神经网的结构。研究快速有效的机器学习方法是提高并行搜索性能的关键．

3. 并行搜索的性能予测

①最好优先搜索和 $\alpha-\beta$ 搜索都属于探试搜索．

为了取得接近线性的加速，在设计一个并行智能机之前，必须予测并行搜索的性能。目前对组合判定问题的并行搜索常采用计算机模拟的方法估计性能。由于在模拟机上运行的问题规模很小，这种小问题的模拟很难反映并行搜索的真实性能，因为随着问题规模增大，组合搜索问题的并行性一般会明显增加。模拟实验需要与理论分析相结合。在模拟与分析时都应特别注意通信对性能的影响。

4. 并行粒度的选择

并行粒度与系统提供的通信能力有关，但更主要的是取决于求解的问题本身。建立一套系统的方法分析智能问题固有的并行性十分必要。在分析并行性时需要考虑启发知识，避免盲目的并行和盲目的搜索。

5. 确定并行选择的策略

选择启发函数值最好的子问题先做并不是必须遵从的原则。在并行条件下，由于全局性的选择开销很大，严格的最好优先在很多场合不适用。我们面临的一个重要课题是如何按选择对性能的影响程度区分各类问题，从而系统地确定合适的选择策略。

6. 并行的淘汰策略

习惯于确定型算法的学者往往忽视淘汰在并行搜索中的作用。实际上选取好的淘汰策略可以大大降低计算复杂性。对于组合判定问题怎样确定淘汰原则至今还是未解决的问题。实现淘汰必然需要通信。在确定淘汰策略时既要考虑通信开销又要考虑通信带来的好处。我们应研究这两者之间的折衷原则。

7. 并行搜索与新技术的结合

近年来光学全息技术开始用于解决人工智能的某些应用问题。全息技术也可看成是一种形式的搜索。如何将并行搜索技术与光学全息技术结合起来是研制光学智能计算机必须考虑的问题。

参 考 文 献

[1] Wah B W, Li Guo-jie, Yu C F. Multiprocessing of combinatorial Search Problems, *Computer*, 18(1985),6,93-108

[2] Reingold E M, Nievergelt J, Deo N. *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977

[3] Horowitz E, Zorat A. Divide and Conquer for Parallel Processing, *IEEE Trans. on Computers*, C-32(1983), 6, 582-585

[4] Burton F W, Huntbach M M. Virtual Tree Machines, *IEEE Trans. on Computers*, C-33(1984), 3,278-280

[5] Li Guo-Jie, Wah B W. Optimal Granularity of Parallel Evaluation of AND Trees, *Proc.* 1986 *Fall Joint Computer Conference*, 297-306

[6] Quinn M J. *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987

[7] Horowitz E, Sahni S, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD,

1978

[8] Ibaraki T. Theoretical Comparisons of Search Strategies in Branch–and–Bound Algorithms, *International Journal of Computer and Information Science*, 5(1976), 4, 315–344

[9] 朱洪，陈增武，段振华，周克成. 算法设计和分析，上海科学技术文献出版社，1989

[10] Mohan J. Experience with Two Parallel Programs Solving the Traveling Salesman Problem, *Proceedings of the 1983 International Conference on Parallel Processing*. 191–193

[11] Lai T–H, Sahni S. Anomalies of Parallel Branch–and–Bound Algorithms, *Communications of the ACM*, 27(1984), 6, 594–602

[12] Marsland T A, Campbell M. Parallel Search of Strongly Ordered Game Trees, *Computing Surveys*, 14(1982),4,533–551

[13] Akl S G, Barnard D, Doran R. Design, Analysis, and Implementation of a Parallel Tree Search Algorithm, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI–4(1982),2, 192–203

[14] Finkel R, Fishburn J. Parallelism in Alpha–Beta Search, *Artificial Intelligence*, 19(1982), 89–106

[15] Marsland T A, Popowich F. Parallel Game–Tree Search, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI–7(1985),4, 442–452

[16] Newborn M. Unsynchronized Iteratively Deepening Parallel Alpha–Beta Search, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI–10(1988), 5, 687–694

[17] Nau D S. An Investigation of the Causes of Pathology in Games, *Artificial Intelligence*, 19(1982), 257–278

[18] Li G–J, Wah B W. MANIP–2: A Multicomputer Architecture for Evaluating Logic Programs, *Proceedings of the 1985 International Conference on Parallel Processing*, 123–130

[19] Das S K, Deo N. Divide–and–Conquer–Based Optimal Parallel Algorithms for Some Graph Problem on EREW PRAM Model, *IEEE Trans. on Circuits and Systems*, 35(1988), 3, 312–322

[20] Das S K, Deo N. Notes on "Divide–and–Conquer–Based Optimal Parallel Algorithms for Some Graph Problems on EREW PRAM Model," *IEEE Trans. on Circuits and Systems*, 37(1990), 7, 962–965

[21] Li Guo–jie, Wah B W, Systolic Processing for Dynamic Programming Problems, *Proceedings of the 1985 International Conference on Parallel Processing*, 434–441

[22] Li Guo–Jie, Wah B W. Computational Efficiency of Parallel Combinatorial OR–Tree Searches, *IEEE Trans. on Software Engineering*, 16(1990),1,13–31

[23] Quinn M. Analysis and Implementation of Branch–and–Bound Algorithms on Hypercube Multicomputer, *IEEE Trans. on Computer*, C–39(1990),3,384–387

[24] Yu C F, Wah B W. Efficient Branch–and–Bound Algorithms on a Two–Level Memory System, *IEEE Trans. on Software Engineering*, 14(1988),9,1342–1356

[25] 高庆狮主编. 智能技术与系统基础（李国杰. 并行组合搜索）. 北京大学出版社，1990

第十四章　神经网络在图论问题中的应用

进入八十年代以来，研究以非线性大规模并行分布式处理为主流的神经网络，取得了引人注目的进展[1,12]．神经网络的应用已经渗透到各个领域，并在智能控制、模式识别、计算机视觉、自适应滤波和信号处理、非线性优化、自动目标识别、连续语音识别、声纳信号的处理、知识处理、传感技术与机器人、生物医学工程等方面取得了令人鼓舞的成就．对神经网络的研究，除了脑神经科学家外，计算机科学、控制论、信息科学、微电子学、心理学、认知科学、物理学和数学等学科的科学家以及各国的企业家也激起了巨大热情和广泛兴趣．人们普遍认为：它将使电子科学、信息科学等领域产生革命性变革，并将促使以神经计算机为基础的高技术群的诞生和发展．

14.1　概　　述

神经网络是一门多学科、综合性的研究领域，引起了各界专家的普遍关注．神经网络的具体形态各异、且还在发展中，但可概括地定义为：由大量简单元件（神经元，可用电子元件、光学元件等模拟）广泛相互连接而成的复杂网络系统，它是在现代神经科学研究成果的基础上提出的，反应了人脑功能的若干基本特征，但并非神经系统的真实写照，而只是其简化、抽象和模拟．即人工神经网络是一种抽象的数学模型，从不同的研究角度和目标出发，它可用作计算模型，或大脑结构模型、或认知模型．本章将它作为计算模型来使用．

研究这一系统的根本目的在于探索人脑加工、贮存和搜索信息的机能，进而探索将此原理应用到各种人工智能的可能性．这也正是此研究具有生命力的根本原因．

14.1.1　生物神经元模型

神经网络的基本形态来源于大脑皮层结构．神经科学研究表明，大脑皮层由大量（约 $10^{11}\sim10^{12}$ 个）神经元组成，虽然可以划分成多种类型的神经元，但它们的基本结构是相似的．每个细胞体有大量树突（Dendrite，即输入端）和轴突（Axon 即输出端）．一个神经元的轴突与另一个神经元的树突的结合部称为突触 (Synapse)．它决定了神经元之间的连接强度和性质（兴奋或抑制），即是决定神经元之间相互作用的强弱和正负．每一个神经元可以有 $10^{1}\sim10^{5}$ 个突触．这就表明大脑皮层是一个广泛连接的复杂网络系统．

生物控制论告诉我们：神经元作为控制和信息处理的基本单元，具有某些重要的功能和特性．比如：具有时空整合的输入信息处理功能；具有兴奋和抑制两种常规工作状态；

突触界面具有脉冲 / 电位信号转换功能；神经纤维传导速度有快有慢；具有突触延时和不应期；具有学习功能、遗忘或疲劳功能（饱和效应）．

随着脑科学和生物控制论研究的进展，人们对神经元的结构和功能有了进一步的了解，神经元并不是一个简单的双稳态逻辑元件，而是超级的微型生物信息处理机。

14.1.2 神经网络的基本特征

神经网络是由大量处理元件互连而成的．网络的信息处理由神经元之间的相互作用来实现；知识与信息的存贮表现为网络元件互连间分布式的物理联系上；网络的学习和识别决定于各神经元连接权的动态演化过程。神经网计算机就是试图模拟这一信息处理机制的一种新型计算机模型，其核心由类似于人脑神经元的简单处理器组成，而处理器之间的联结则与神经元之间的突触联系相似。

1. 神经元的形式化描述

神经元是神经网络的基本处理单元，它一般是一个多输入 / 单输出的非线性器件，其结构模型如图 14.1 所示。其中：u_i 为神经元的内部状态，θ_i 为阀值，x_i 为输入信号，w_{ij} 表示从 u_j 到 u_i 连接的权值，s_i 表示外部输入信号（在某些情况下，它可以控制神经元 u_i，使得它保持在某一状态）。这种结构模型可形式地描述为：

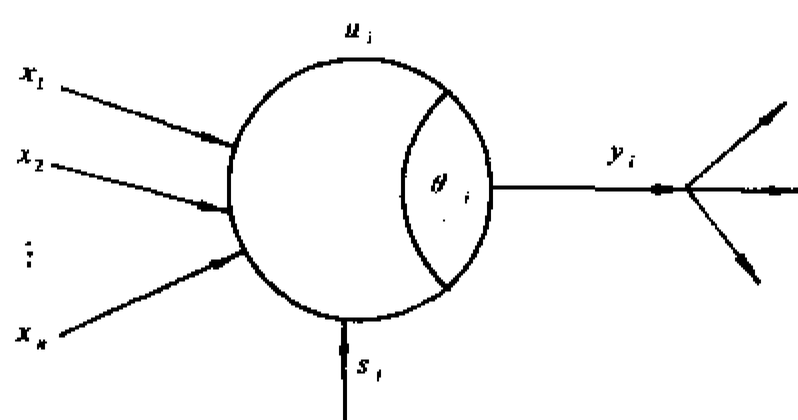


图 14.1　神经元结构模型

$$\sigma_i = \sum_j w_{ij} x_j + s_i - \theta_i$$

$$u_i = f(\sigma_i)$$

$$y_i = g(u_i) = h(\sigma_i), \qquad h = g \cdot f$$

当神经元没有内部状态时，可以令 $f = I$（恒等映射），如图 14.2 所示。常用的神经元非线性特性可描述如下：

(1)　阀值型：在这种模型中，神经元没有内部状态，而且函数 f 是一个阶跃函数，（见图 14.2)(a)），即：

$$h(x_i) = f(x_i) = u(x_i)$$

$$f(x_i) = \begin{cases} 1, & x_i > 0 \\ 0, & x_i \leqslant 0 \end{cases}$$

这是最早提出的二值离散神经元模型；

(2)　分段线性型，如图14.2(b)所示；

(3)　S 状：它一般是没有内部状态并且连续取值，其 I / O 特性常用对数或双曲正切等一类 S 曲线来表示，如 $x_i = 1/(1+\exp(\sigma_i))$ 或 $x_i = [1+\tanh(\sigma_i / x_0)]/2$ 等，这类曲线反映了神经元的饱和特性。

2. 神经网络模型

按照人工神经网络对生物神经系统的不同组织层次和抽象层次的模拟，神经网络模型可分为下列五类：

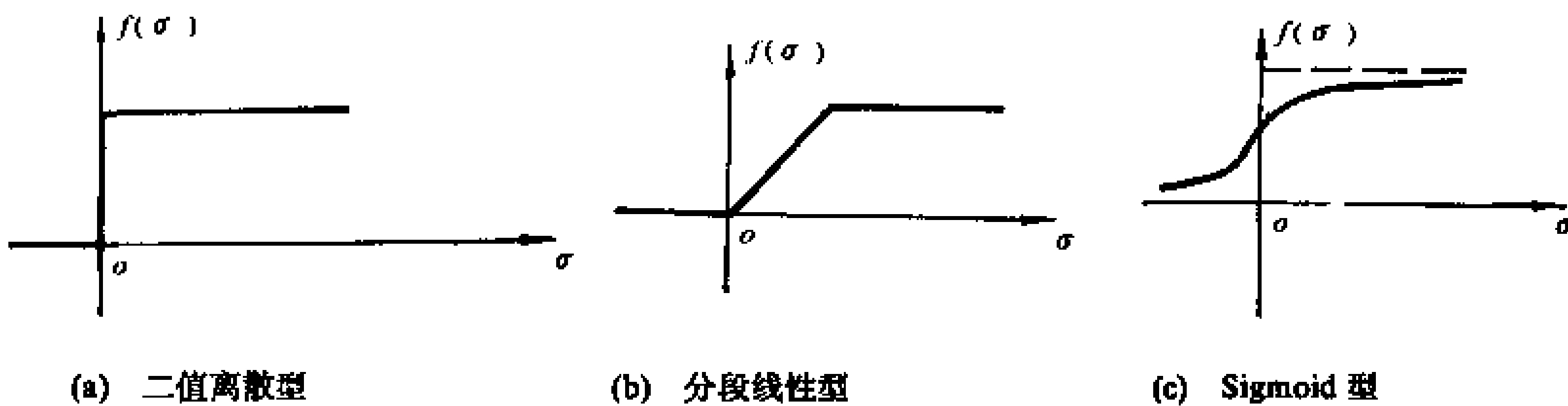
(1) 神经元层次模型：主要研究单个神经元的动态特性和自适应性，探索神经元对输入信息有选择的响应和某些基本存贮功能的机理；

(2) 组合式模型：它由数种相互补充、相互协作的神经元组成，用于完成某些特定的任务，如模式识别，机器人控制等；

(3) 网络层次模型：它是由许多相同神经元连接组成的网络，从整体上研究网络的集体特性，如 Hopfield 神经网络[7]；

(4) 神经系统层次模型：一般由多个不同性质的神经网络构成，以模拟生物神经的更复杂更抽象的性质，如自动识别、概念形成、全局稳定控制等；

(5) 智能型模型：这是最抽象的层次模型，大多数以语言形式模拟人脑信息处理的运行、过程、算法和策略。这类模型试图模拟诸如感知、思维、问题求解等基本过程，而且和人工智能紧密相关。



(a) 二值离散型　　(b) 分段线性型　　(c) Sigmoid 型

图 14.2 神经元的 I／O 特性

神经网络模型很多，而且现有模型远远不是完备和成熟的，有待进一步发展。总体来说，它应力图体现人脑的基本特征。因而神经网络具有以下的基本特征：

① 以大规模模拟并行处理为主。这里的并行处理决不是简单的“以空间的复杂性代替时间复杂性”，而是反映了不同的“计算”原理。因此，不能将神经网络的大规模并行处理与多处理器的并行机等同起来。单纯的并行机制不能很好地体现因果关系和信息的相互影响，功能必然过于简单。好的网络应是大规模并行处理和串行处理的有机结合。关于模拟计算，历史上曾因它的计算精度太低而被数值计算所代替，神经网络则不同。它擅长的并不是精密的数值计算或绝对准确的决策，只是要求基本正确。正如人不可能绝对正确一样；神经网络主要涉及有限次“运算”，而且它本身具有容错性和纠错性，误差不会积累，高精度实际上并无必要。在这里，模拟计算对减少计算工作量起到了重要的作用。

②具有很强的鲁棒性和容错性，善于联想、概括、类比和推广。由于信息存贮本质上是分布式的，任何局部的损伤不会影响整体的结果。

③具有很强的自学习能力。系统可以在学习过程中不断完善自己，具有创新特点，这不同于人工智能中的专家系统，后者只是专家经验的知识库，并不能创新和发展。

④它是一个具有高度非线性的超大规模连续时间动力系统，具有集体运算的能

力和一般非线性动力系统的共性，即不可预测性、吸引性、耗散性、非平衡性，不可逆性，广泛联结性和自适应性等．这同本质上是线性系统的传统数字计算机迥然不同。

3. 神经网络的信息处理能力

神经网络的信息处理能力包括:①网络的信息存贮能力;②网络的计算能力。它们对应下面的问题:

(i) 在一个含 N 个神经元的神经网络中，可存贮多少信息?

(ii) 神经网络具有什么样的计算能力，即它能够有效地计算哪些问题?

存贮能力与计算能力是现代计算机科学的两个基本问题，在人工神经网络中，它们同样构成了神经网络理论的两个最基本的问题。

在传统的数字计算机中，计算与存贮是完全独立的两个部分，即计算机在计算前先要从存贮器中取出待处理的数据，然后再进行计算，最后又将计算结果放入存贮器中。这样，存贮器与运算器之间的通道就形成了现代计算机的瓶颈，从而大大限制了计算机的计算能力。

在人工神经网络系统中，信息的存贮与处理（计算）是合为一体的，即信息的存贮体现在神经元互连的分布上，并以大规模并行分布方式处理。从系统论的观点看，可以把神经网络看成是由大量子系统组成的大系统，系统的最终行为完全由它的吸引子决定．如果将动力系统的吸引子视作记忆的话，那么从初态向吸引子的流动过程就是寻找记忆的过程．初态可以看作是给定有记忆的部分信息．换句话说，流动的过程就是从部分信息找出全部信息的过程，这就是联想记忆的基本原理．进一步说，若把动力系统的稳定吸引子看成为系统计算能量函数的极小点，则系统最终会流向期望的最小点，"计算"也就在运动过程中悄悄地完成了，运动的时间就是计算的时间，这就是神经网络计算机的基本原理，即利用吸引子进行存贮和计算。

从广义角度讲，微积分，文字翻译、推理等都是一个计算过程．而从数学观点看，计算就是在满足一定公理、定理的条件下，从一空间到另一空间的代数映射；从物理观点看，计算是按照一定的自然规则，在某种"硬件"上所发生的一些物理规则．因此，计算可以表示为一动力系统中的状态空间变换的轨迹。神经网络的计算就是其中状态的转换，其计算过程可看成是状态的转换过程，对给定的输入，它的计算结果是系统的稳定状态。

目前已开发的 30 多种神经网络模型都是针对某种特殊用途的，因而对这些特殊问题有很强的计算能力。如何在具体模型中更好地体现神经网络的特征，仍有大量的工作要做．而且，传统的途径在许多问题上还是很有效的。现在，人们继续努力寻找新的机制，以构造通用的神经网络模型，

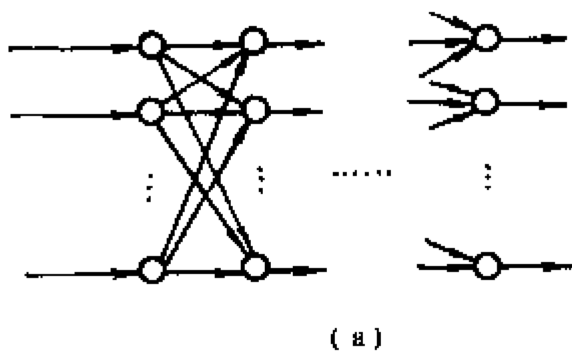
迄今为止，人们发现神经网络可以完成的信息处理任务有：数学逼近映射；概率密度函数的估计；从二进制数据基中提取相关的知识；形成拓扑连续及统计意义上的同构映射；最近相邻模式分类；数据聚集；最优化问题的计算等．

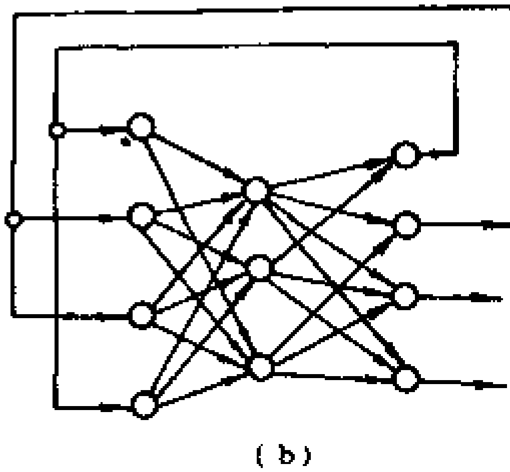
4. 神经网络的互连结构形态

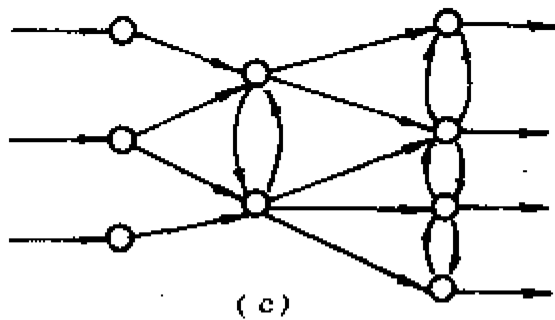
根据连接方式的不同，神经网络可分成以下几种类型:

(1) 无反馈的前向网络．如图 14.3(a)所示，神经元分层排列，组成输入层、隐层（中间层）和输出层。每一层的神经元只接受前一层神经元的输入。输入模式经过各

层的顺次变换后，得到输出层的输出[13]。


(a)


(b)


(c)

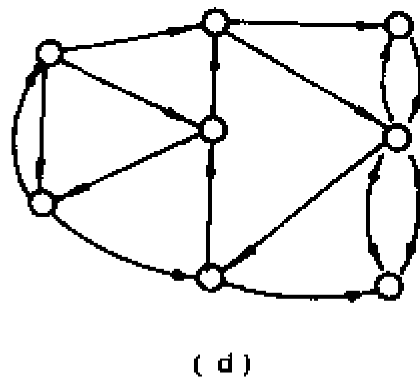

(d)

图 14.3 网络结构的各种形态

(2) 有反馈的前向网络。如图 14.3(b)所示，从输出层到输入层含有反馈。

(3) 层内有相互结合的前向网络。如图 14.3(c)所示，通过层内神经元间的相互结合，可以实现同一层内神经元之间的横向抑制或兴奋机制，这样可以限制每层内能同时动作的神经元数目，或者把每层内的神经元分为若干组，让每组作为一个整体来动作。

(4) 相互结合型网络。如图 14.3(d)所示，这种网络在任意两个神经元之间都可能有连接。在无反馈的前向网络中，信号一旦通过某个神经元，过程就结束了；而在相互结合网络中，信号要在神经元之间反复往返传递，网络处在一种不断改变状态的动态过程中。从某一初态开始，经过若干次的变化，才会到达某种平衡状态，根据网络的结构和神经元的特性，还有可能进入周期振荡或其它如浑沌等平衡状态[7]。

(1)、(2)、(3)可看作是(4)的一种特殊情况，但不论从网络的计算和学习机制看，还是从它们的应用场合看，都有很大的区别。

5. 神经网络的工作方式

神经网络的工作过程主要由两个阶段组成。一个阶段是工作期，此时各连接权值固定，计算单元的状态逐渐演化，以求达到稳定状态。另一阶段是学习期（自适应期，或设计期），此时各计算单元状态不变，各连接权值可以修改（通过学习样本或其它方法）。前一阶段较快，各单元的状态亦称短期记忆，后一阶段慢得多，权及连接方式亦称长期记忆。

能量函数是神经网络的一个基本量。按对能量函数的利用可分为三种工作方式：

(1) 能量函数的所有局部极小点都起作用。这一类主要用于各种联想存贮器，信息压缩及编码。

(2) 只利用能量函数的全局最小点。这一类主要用于求解组合最优化问题。

(3) 在工作中不考虑能量函数，主要作用是函数映射，它主要用于模式分类和特征抽取。

6. 神经网络的学习规则

计算机的机器学习分为三类：死记式学习、从例子中学习和无导师学习。神经网络的学习亦可作类似的分类。例如一类网络事先设计成记忆特殊的模式，以后当给定有关该系统的输入信息时，它们就被回忆起来。Hopfield 模型[7]，即是这类死记式学习的例子。而

许多网络则是从例子中学习[13]，在学习时往往先给网络提供一个输入模式，通过期望输出的最佳估计，网络对它作出响应，然后教师给出正确的输出模式。若有必要，则允许系统调节权值，使得内部表示更接近期望的结果，感知器就是这种教师学习的例子。另一类网络则设计成不需要教师直接指点的学习方式，如竞争学习系统[11]。其学习过程为：给系统提供的动态输入信息流，使得各个单元成为具有不同输入特性的特征检测器，从而将事件空间分为有用的多个区域。自适应共振网络[6]就是无教师学习的例子。

学习规则可以分为三类，即

1）相关规则：仅仅根据连接间的激活水平改变权矩阵；

2）纠错规则：依赖于输出结点的外部反馈改变权矩阵；

3）无教师学习规则：学习表现为自适应于输入空间的检测规则。

在人工神经网络中，学习规则是修正权的一个算法，以便获得合适的映射函数或其它系统性能。Hebb 学习①的相关假设是许多规则的基础（尤其是相关规则）；Hopfield 神经网络和自组织特征映射展示了有效的模式识别能力；纠错规则常使用梯度下降法，它存在局部极小问题；无教师学习规则提供了新的选择，它利用自适应学习方法，使结点有选择地接收输入空间上的不同特性，从而抛弃了普通神经网络学习映射函数的学习概念，并提供了基于检测特性空间的活动规律的性能描述。

14.2 Hopfield 模型和旅行商问题

14.2.1 引　言

组合优化产生于现实世界中寻找最优值的问题，它的目标是求解反映客观世界的多变量函数的极值。由于问题本身的复杂性，组合优化中的很多问题往往是 NP－完全的，象旅行商问题，图划分问题等。因此人们只好用能在多项式时间内求解的启发式算法来寻找满意解以满足实际工作中的需要。但启发式算法存在着以下不足：首先它与问题特例有关。如针对某一特定领域设计的高效算法对其它问题可能会失效。其次用启发式算法求解组合优化问题一般是在软件层次上开展工作，与飞速发展的超大规模集成电路（VLSI）的联系不是十分密切。再者以搜索为基础的启发式算法与人脑处理问题的模式有着较大的差距。所以人们在不断完善已有的算法基础上，一直在寻找新的计算模型以更加有效地解决客观世界中的优化问题。

Hopfield[7] 开创性的在物理学、神经生物学、计算机科学等领域间架起了桥梁。他提出并证明了：在高强度连接下的神经网依靠集体协同作用能自发产生计算行为这一科学论断，开辟了神经网模型在计算机科学应用中的新天地；开创了一条组合优化的新途径。随后，Kirkpatrick 等人[10] 提出了“模拟退火”法，Durbin 等人[4] 提出了“弹性网”法，Kohonen[11] 提出了“自组织映射”方法等可用于组合优化的计算模型。用神经网来进行组合优化处理日益受到人们的重视。

---

①$\Delta w_{ij}=aS_iS_j$，$a>0$，若i，j神经元同时兴奋（$S_i=S_j=1$），则它们之间的连接应加强。

由于 Hopfield – Tank 模型（以下简称 HT 模型）影响最大，时间性能上最优，本节以它为重点展开讨论，其他模型留在下节中讲述。在第二小节中分别介绍 Hopfield 离散和连续两种模型，通过对神经元的输入输出关系，运动方程和网络的能量函数这三个方面的数学描述，揭示了 Hopfield 神经网就是极小值求解机这一计算特征。在第三小节结合旅行商问题（TSP），阐述能量函数在 HT 模型中的重要作用。在第四小节中提出了一个基于邻接矩阵的能量函数，它具有运算速度快、解的质量高，便于 VLSI 实现等优点，是对 HT 模型的一个较大的改进。

14.2.2 Hopfield模型简介

我们将从神经元的输入输出关系，运动方程和网络的能量函数这三个数学概念出发来描述 Hopfield 神经网络。

1. 离散异步 Hopfield 模型

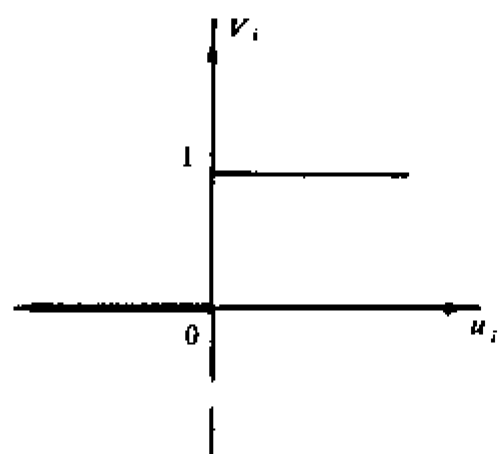
离散异步的 Hopfield 模型是最早提出的，它与统计物理学中的自旋玻璃态有着对应关系，是以后要介绍的连续确定型模型的基础。

离散模型中每个神经元的输入 u_i 和输出 V_i 满足阶梯函数:

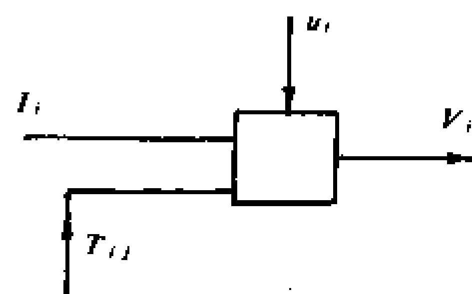
$$V_i = \mathrm{step}(u_i) \tag{14.2.1}$$

不妨假定 V_i 只取 0 和 1 两个值，则其输入输出关系见图 14.4(*a*)。

每个神经元 i 与其它神经元 j 都有联系，用连接强度 T_{ij} 表示第 j 个神经元对第 i 个神经元的输入连接强度，它类似于真实神经元的突触连接强度。神经元 i 的输入有两类：一是外部输入 I_i；另一来自其它神经元。因此神经元 i 的总输入是：$\sum_{i\neq j} T_{ij} V_j + I_i$。



(a) 0／1 输入输出关系



(b) 判断电路

图 14.4 离散神经元

每个神经元以平均速度 W_i 随机地改变其状态，变化规律遵循如下的运动方程:

$$V_i = \begin{cases} 1, & \sum_{i\neq j} T_{ij} V_j + I_i \geqslant u_i \\ 0, & \sum_{i\neq j} T_{ij} V_j + I_i < u_i \end{cases} \tag{14.2.2}$$

其中 u_i 是每个神经元的阈值，见图 14.4(*b*)。

如果神经网的连接矩阵 T 是对称的且对角线元素为零，构造如下的能量函数：

$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} T_{ij} V_i V_j - \sum_i I_i V_i + \sum_i u_i V_i \tag{14.2.3}$$

我们考察由于某个神经元 i 的改变 ΔV_i 而引起的 E 的的变化 ΔE：

$$\Delta E = -\left[\sum_{i\neq j} T_{ij} V_j + I_i - u_i\right]\Delta V_i$$

方括号内恰好是方程(14.2.2)右端的判别条件，它和ΔV_i同号，所以：$\Delta E \leqslant 0$。

由于能量函数 E 是有界的，这种随机异步的变化过程必然趋向稳定状态，将状态固定在 N 维超立方体的顶点上，这说明上述神经网络能对（14.2.3）式的函数进行极小值运算．即 Hopfield 离散模型的计算函数 H 是：

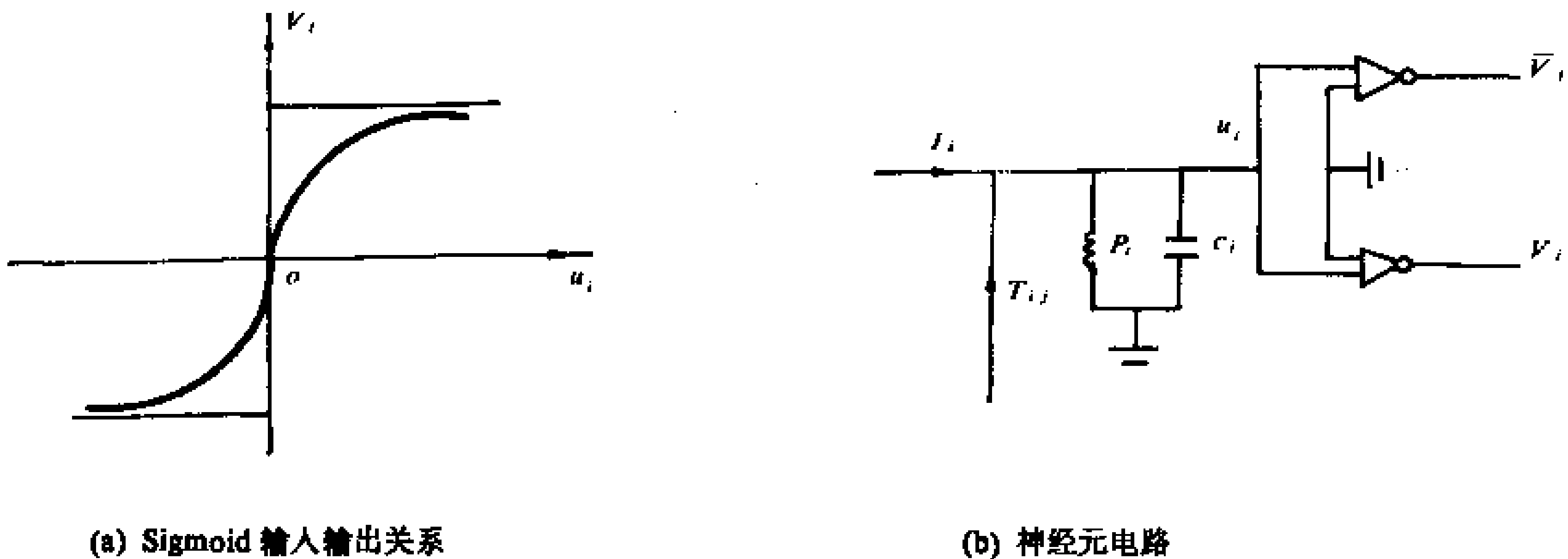
$$H(E, \vec{V}_0) = \min(E, \vec{V}_0) \tag{14.2.4}$$

其中$\vec{V}_0 = (V_1^0,\ V_2^0,\ \cdots,\ V_n^0)$是某一初始状态．

因此，如果能把优化问题映射成 (14.2.3) 式的能量函数形式，即么就可用 Hopfield 模型自动进行优化处理了．

2. 连续确定型模型

离散模型中的神经元与真实神经元，甚至与简单的电路器件相比差别很大．这主要表现在：第一，真实神经元的输入输出关系是连续的．第二，真实神经元由于存在着时间延迟，其运动方程应由微分方程刻化．为此，Hopfield[8] 提出了连续确定型模型以更加逼近真实神经元，并有利于 VLSI 的硬件实现和改善网络的总体性能．



(a) Sigmoid 输入输出关系　　(b) 神经元电路

图 14.5　连续神经元

连续型神经元的输入输出关系是一个单调递增的 Sigmoid 函数，见图 14.5(a)．其方程抽象如下：

$$V_i = g_i(u_i) \tag{14.2.5}$$

每个神经元的电路图如图 14.5(b) 所示．图中 I_i 是刺激电流，用于维持整个网络的活跃状态．ρ_i 和 C_i 分别是放大器的输入电阻和总电容．T_{ij} 是第 j 个神经元和第 i 个神经元的电导，即 $|T_{ij}| = 1/R_{ij}$，V_j 的两个输出要视 T_{ij} 的正负分别联到 V_i 的输入

端。所以神经元的RC方程是:

$$C_i(\mathrm{d}u_i / \mathrm{d}t) = \sum_j T_{ij} V_j - u_i / R_i + I_i \tag{14.2.6}$$

其中: $1 / R_i = 1 / \rho_i + \sum_j 1 / R_{ij}$.

考虑如下Lyapunov能量函数:

$$E = -\frac{1}{2}\sum_i\sum_j T_{ij} V_i V_j + \sum_i \frac{1}{R_i}\int_0^{V_i} g_i^{-1}(V)\mathrm{d}V - \sum_i I_i V_i \tag{14.2.7}$$

它关于对称矩阵T的时间导数是:

$$\begin{aligned}\frac{\mathrm{d}E}{\mathrm{d}t} &= -\sum_i \mathrm{d}V_i / \mathrm{d}t(\sum_j T_{ij} V_j - u_i / R_i + I_i) \\ &= -\sum_i C_i(\mathrm{d}u_i / \mathrm{d}t) \cdot \mathrm{d}V_i / \mathrm{d}t \\ &= -\sum_i C_i g_i^{-1}(V_i)(\mathrm{d}V_i / \mathrm{d}t)^2\end{aligned}$$

因为 g_i 是单调递增函数而 C_i 又大于零，所以每项和非负．故得到 $\mathrm{d}E / \mathrm{d}t \leqslant 0$.

当 $\mathrm{d}E / \mathrm{d}t = 0$ 时，可得 $\mathrm{d}V_i / \mathrm{d}t = 0$，对所有 i 成立，即系统达到稳定状态．

这说明具有式 (14.2.7) 的系统的演变过程就是在 $[0,1]^n$ 空间内寻找极小值稳定点（吸引子）的过程，并在达到这些点后稳定下来．因此这种神经网络同样具有自动求极小值的计算功能．

而离散模型和连续模型的对应关系需借助放大系数来说明．如果令 $V_i = g_i(\lambda u_i)$，其中 λ 表示放大系数．当 λ 很大时 ($\lambda \to \infty$)，离散模型的稳定点与连续模型的稳定点是对应的．当 λ 有限时，极小值点位于 $[0,1]^n$ 空间的内部．因此在实际应用中一般取较大的 λ.

14.2.3 HT模型下的旅行商问题

Hopfield 在提出连续确定型模型之后与 Tank 合作，成功地用 HT 模型对旅行商 (TSP) 这一经典的组合优化难题给出了一个满意解，为神经网解决组合优化问题开辟了一条新途径[9]．随后人们将其应用到模拟电路，线性规划，图论和作业分配等许多领域中[14,21]．

HT 模型是 Hopfield 连续模型的应用．下面结合 TSP 问题的求解，介绍一下用 HT 模型处理问题的一般方法．

1. 问题的表示

选择合适的表达方法使神经元的稳定输出对应于问题的解答．

我们知道 TSP 问题是寻找一条最短的周游 n 个城市的路径．如果 $V_{Xi} \in [0,1]$，当网络稳定时，用 $V_{Xi} = 1$ 表示第 X 个城市在周游中第 i 次被走到，那么 X 城市的向量: (0,0,0,1,0) 表示它第 4 次被走到．而 5 个城市的一条周游路径 BDEAC 表示成如下形

式的置换矩阵:

	1	2	3	4	5
A	0	0	0	1	0
B	1	0	0	0	0
C	0	0	0	0	1
D	0	1	0	0	0
E	0	0	1	0	0

因为每个城市仅允许访问一次，因此矩阵中的每行（列）只有一个 1，总共有 n 个 1。

2. **能量函数的选取**

选择合适的能量函数使其最小值对应于问题的最优解。

这里首先要保证极小值是合法的，即当能量函数取极小值时问题的约束条件被满足。对于上面所述的置换矩阵的约束条件可描述如下:

$$E_{约束}=\sum_X\sum_i\sum_{j\neq i}V_{Xi}V_{Xj}+\sum_i\sum_X\sum_{Y\neq X}V_{Xi}V_{Yi}+(\sum_X\sum_i V_{Xi}-N)^2$$

其中各项分别表示每行（列）只有一个 1，总共有N个 1 时$E_{约束}$取最小值零。

然后构造目标函数使其满足约束条件下的解对应于问题的一个满意的解. 对于 TSP 问题，可构造如下:

$$E_{目标}=\sum_X\sum_{Y\neq X}\sum_i d_{XY}V_{Xi}(V_{Y,i+1}+V_{Y,i-1})$$

其中对 i 下标取模运算表示循环路径。

上述公式表明：如果边 $X\to Y$ 是一段路径，那么矩阵中相邻两列 i 和 $i+1$(或 $i-1$) 必有两个 1 表示相应的两个城市先后被走过. 当约束条件满足时，$E_{目标}$就是周游路径的总长，因此，总能量函数是:

$$E=\frac{A}{2}\sum_X\sum_i\sum_{j\neq i}V_{Xi}V_{Xj}+\frac{B}{2}\sum_i\sum_X\sum_{Y\neq X}V_{Xi}V_{Yi}+\frac{C}{2}(\sum_X\sum_i V_{Xi}-N)^2$$
$$+\frac{D}{2}\sum_X\sum_{Y\neq X}\sum_i d_{XY}V_{Xi}(V_{Y,i+1}+V_{Y,i-1}) \tag{14.2.8}$$

它由约束条件和目标函数两部份组成. 其中各项的系数是为了以后化简的方便。

3. **模拟神经元的运动方程**

对于式(14.2.8)的运动方程是:

$$\mathrm{d}u_{Xi}/\mathrm{d}t=-\frac{u_{Xi}}{\tau}-\frac{\partial E}{\partial V_{Xi}} \tag{14.2.9}$$

其中 $\tau=RC$ 是时间常数，u_{Xi}/τ 的存在是原能量函数中的积分项所致. 展开式 (14.2.9) 可得:

$$\mathrm{d}u_{X_i}/\mathrm{d}t=-\frac{u_{Xi}}{\tau}-A\sum_{j\neq i}V_{Xj}-B\sum_{Y\neq X}V_{Yi}-C(\sum_Y\sum_j V_{Yj}-N)$$

$$-D\sum_{Y} d_{XY}(V_{Y,i+1}+V_{Y,i-1}) \tag{14.2.10}$$

在 HT 模型中，神经元的输入输出取自如下函数:

$$V_{Xi}=g(u_{Xi})=(1+\tanh(\lambda u_{Xi}))/2 \tag{14.2.11}$$

这样上述三式就完全刻划了 TSP 问题的 HT 模型。如果用软件模拟，可采用某一数值方法积分方程 (14.2.10)，其算法复杂度是 $O(N^3)$；如果用硬件实现，就需知道连接矩阵 T。由于连接系数 T_{ij} 是二次项的系数 (见 14.2.7)，所以 $T_{ij}=-\partial E/\partial V_i \partial V_j$，即

$$T_{Xi,Yj}=-A\delta_{XY}(1-\delta_{ij})-B\delta_{ij}(1-\delta_{XY})-C-Dd_{XY}(\delta_{j,i+1}+\delta_{j,i-1})$$

其中

$$\delta_{ij}=\begin{cases}1, & i=j \text{ 时}\\ 0, & i\neq j \text{ 时}\end{cases}$$

对此问题在单机上模拟实现的时间复杂度是 $O(N^4)$，故在模拟上一般以神经元的运动方程为准。

4. **初值选择**

选一组初值 u^0_{Xi} 和网络参数 A、B、C、D 等，运行该网络以求得解答。

正如前面所述的，HT模型与初始值u^0_{Xi}有关。在TSP问题中原则上可选 $u^0_{Xi}=1/N$ 为每个神经元的初始值，因为它满足$\sum_X\sum_i u^0_{Xi}=N$。但由于相同长度的等价路径有 $2N$ 条，系统无法从中决择，所以要加一定的燥声值（$\pm 0.1u^0_{Xi}$）来打破这种平衡。关于各种参数的具体值见文 [9]，有关模拟过程见文 [20]。

14.2.4 HT模型的改进

HT 模型虽然在时间性能上有着较大的优越性，但在实验中发现网络存在着不稳定性和网络参数的敏感性等缺陷。特别是 Wilson[20] 指出：在城市数目 $N=10$ 时只有 8% 是有效解和找不到 $N=64$ 的网络参数后这个问题显得尤其尖锐。问题出现在最后阶段，但毛病的根源来源于前三步，为此展开了大量的研究工作。

在问题表达方面，Brandt[2] 提出了一个三下标表示方法。即 V_{ijk} 为 1 表示从第 i 个城市到第 j 个城市第 k 次被走到。这样路径总长就是：$\sum_i\sum_j\sum_k d_{ij}\ V_{ijk}$。其优点是：数据项 d_{ij} 仅是能量函数中一次项的系数，它与 T_{ij} 无关，有利于硬件电路的实现。但它需要 $O(n^3)$ 个神经元，计算量很大。

在能量函数方面，Szu[15] 和 Brandt[2] 等提出了一个约束加强的能量公式:

$$E_{约束}=\sum_X(\sum_i V_{Xi}-1)^2+\sum_i(\sum_X V_{xi}-1)^2 \tag{14.2.11}$$

此式表明每行每列有且仅有一个 1，这种约束能及时唤醒位于全零元素行（列）中的神经

元，抑制过多的 1 出现。

虽然上述表示方法提高了收敛性，但由于目标函数未加改变，所以解的质量不是很高。

对于神经元的运动方程 (14.2.9)，一般采用 Euler 差分法，即:

$$u_{Xi}^{t+\Delta t}=u_{Xi}^{t}\left(1-\frac{\Delta t}{\tau}\right)-\frac{\partial E}{\partial V_{Xi}}\cdot\Delta t \tag{14.2.12}$$

通常采用很小的 Δt 以逼近真实情况。如 Wilson 取 $\Delta t=10^{-5}$，因此模拟的速度很慢。Szu 取 $\Delta t/t=1$，使运动方程成为梯度函数以加快运行速度。事实上，Δt 既不能太大也不能过小，要结合放大系数 λ 综合考虑。若 Δt 太大，则会丢失某些历史信息，使得 $u_i^{t+\Delta t}$ 与 u_i^t 无关，忘记了某些先前的正确决策。若 Δt 太小，则得不到及时的反馈信息，容易无所适从。下面介绍的模型中，控制 $\lambda\Delta t/\tau\approx 1$，以便平衡各方面的因素，可以取得较好的效果。

针对上述能量函数的不足，我们介绍一个以边为主的邻接矩阵公式。当 $V_{Xi}=1$，表示从顶点 X 到顶点 i 的这条边被选中，例如 BDEAC 这条周游路径的邻接矩阵是:

	1	2	3	4	5
A	0	0	1	0	1
B	0	0	1	1	0
C	1	1	0	0	0
D	0	1	0	0	1
E	1	0	0	1	0

其中 $V_{ii}=0$，表示不存在孤立顶点的回路。$V_{ij}=V_{ji}$ 表示边中的两个顶点是对称的。因此这个对称阵的能量函数是:

$$E=\frac{A}{2}\sum_i(\sum_j V_{ij}-2)^2+\frac{B}{2}\sum_j(\sum_i V_{ij}-2)^2+C\sum_i\sum_j V_{ij}(1-V_{ji})$$

$$+D\sum_i\sum_j d_{ij}V_{ij} \tag{14.2.13}$$

其中前面两项表示每行，每列有两个 1，第三项表示对称关系。这个能量函数具有如下优点:

1) 整个网络只需 N^2-N 个神经元，比其同类模型Brandt[2]少了一个数量级。

2) 在 TSP 问题中，任何一个解都有 $2N-1$ 个等长路径与之对应。对称性的引入自动实现了边的双向性，减少了解的等价类数目，提高了网络的计算效率。

3) 神经元的运动方程是:

$$\mathrm{d}u_{ij}/\mathrm{d}t=-u_{ij}/\tau-A(\sum_j V_{ij}-2)-B(\sum_i V_{ij}-2)-C(1-2V_{ji})-Dd_{ij} \tag{14.2.14}$$

其中数据项 d_{ij} 直接施力于方程 (14.2.14)，能产生较优的解。

4) 数据项 d_{ij} 在能量函数中线性出现，使得它仅以偏流的形式出现在输入中，网络的连接矩阵是固定的，不以问题实例的改变而变化，有利于 VLSI 的实现，即

$$I_{Xi}=-2(A+B)+C+D\mathrm{d}_{ij} \tag{14.2.15}$$

$$T_{Xi,Yj} = A\delta_{XY} + B\delta_{ij} - 2C\delta_{Xj}\ \delta_{Yi} \tag{14.2.16}$$

从 (14.2.16) 式中还可以看出：神经元间的连接复杂性已有所降低，从 Hopfield 的全连接变为仅与本行、本列和对称元素连接了。这同样有利于 VLSI 的硬件实现。

实验结果表明：上述模型容易产生非法回路因而导致非法解，这里的非法回路是指顶点个数小于 N 的回路。而这个问题是单层网络所无法解决的，为此引入第二层探测网络来进行非法回路的检查和纠正[17]。

设 A 为图 G 的邻接矩阵，I 为单位矩阵，则令 $B=\sum_i (A+I)^i$，元素 b_{ij} 表示顶点 i 和 j 之间的连通性。如果第一列元素 $b_{i1}=0$，那么顶点 i 不能从顶点 1 到达，即是说所有 $b_{i1}=1$ 的顶点将形成非法回路。所以第二层网络的每个元素 b_{ij} 和第一层的 V_{ij} 相连，通过 $B_{N\times N}$ 矩阵的运算（最多 N 次）就可检查出是否存在着非法回路。一旦找出非法回路，可通过偏流 I_i 的作用来抑制小回路的生成。即是

$$\Delta I_{XY} = \begin{cases} 0, & 当 S_{Xi}=1,\quad \forall x=1,\cdots,N \\ \alpha(d_{max}-d_{XY}), & 当 S_{X1}=1-S_{Y1} \\ -\alpha d_{XY}, & 当 S_{Xi}=S_{Y1} \end{cases}$$

这里 S_{XY} 是第二层的输出。条件 1 表示 X 和 Y 同在一个回路中，条件 2 表示 X 和 Y 不在一个回路中，它们分别属于不同的划分，这时增大刺激电流使它们建立联系。条件 3 则说明抑制同一非法回路中的顶点建立联系。这样，最终所有顶点将和顶点 1 同属一个回路，即是合法的周游路径。

由此可见，引入第二层检查和纠正网络，使 Hopfield 模型增强了活力，具有自学习的功能，提高了问题求解的能力。

14.3 其他模型和旅行商问题

本节以 TSP 问题为例介绍其他神经网模型在优化计算中的应用。一方面通过对各模型的简单介绍，使得对各种模型有个基本的了解；另一方面通过对 TSP 问题的研究，希望能起到举一反三的作用，以便推广应用到其他图论问题的求解。本节所介绍的四种方法两两相似，即弹性网法和自组织映射比较类似；模拟退火和均场退火有共同之处。因此，我们除了介绍这些方法本身外，还同时进行简单的比较，以加深对各模型的掌握和了解。

14.3.1 弹性网法

弹性网法(Elastic Net Method)是 Durbin 和 Willshaw[4]提出的一个并行模拟算法。该算法基于几何观点考虑问题，即一条周游路径是圆周到城市所在平面的映射。我们考虑这样一个映射，它把圆周上的一些点映射到平面上的一组点，使得圆周上的相邻点经映射后尽可能地保持接近。算法的过程就是不断地修改映射过程，即不断地计算在城市平面上映

射像点的位置使它们保持邻近性。由于最初圆周位于城市分布的中心处，它不断地向四周任意扩张，最终通过（或接近）所有城市，从而形成一条周游路径。因此，在 TSP 问题中周游路径的几何约束条件是自然满足的。并且，路径中的每一点在移动过程中受两种力的影响。第一种力使得它向最近的城市运动；第二种力是邻居间的张力又使总的路径最短。这样，一个城市在点的运动中就有可能和路径中的某一段相关而不仅只和一个点对应，这种关系的紧密程度和城市与点、点与点间的距离有关。起初所有的城市对路径中的每点施加大致相同的影响。随后，距离远的影响逐渐减弱，每个城市只和与它最接近的点有关。这种演化过程可由平滑半径参数 K 来控制，使得运动点趋向于城市点。

由此可见，这种算法之所以称为弹性网法，是由于它像一根有弹性的橡皮条从最初状态不断地发生形变，最终导致一条周游路径。

如果用向量 x_i 来表示城市 i 的位置坐标，向量 y_j 表示路径中的某点 j 的位置坐标，那么它的能量函数是：

$$E = -\alpha K\sum_i \ln(\sum_j \varphi(|x_i - y_j|,K)) + \beta\sum_j |y_{j+1} - y_j|^2$$

其中 $\varphi(d,K)$ 是一个正的、有界的、随 d 递减的函数。当 $d > K$ 时，它趋于零。在文 [4] 中，取 $\varphi(d,K) = \exp(-d^2/2K)$。能量函数中的两项分别表示了上述的两种力，而 α 和 β 分别决定了两种力的作用大小。平滑半径 K 刻划了路径上的点与城市的接近程度。在极限情况下，$K\to 0$，为了使得 E 有界，极限路径就必须经过所有的城市（否则，$d\neq 0$，$\varphi(d,K)\to 0, -\ln\sum\varphi(d,K)\to +\infty$）。

y_j 的座标由梯度下降法来决定，即是

$$\Delta y_j = -K\frac{\partial E}{\partial y_j}$$

亦即

$$\Delta y_j = \alpha\sum_i W_{ij}(x_i - y_j) + \beta K(y_{j+1} - 2y_j + y_{j-1})$$

其中

$$W_{ij} = \frac{\varphi(|x_i - y_j|,K)}{\sum_k \varphi(|x_i - y_k|,K)}$$

由此可见，在能量函数中引用自然对数，使得导数运算方便。

由于 y_j 是沿着负梯度方向变化，所以 E 是不断减少的；而 E 是有下界的（最优解），因此上述过程必将到达一个 E 的极小值而稳定。在极限情况下，$K\to 0$，路径点数趋于无穷，最终通过所有城市，并获得能量最小值后达到最优解。这暗示我们：使用弹性网算法，即使不在极限状态下也能得到较好的解。

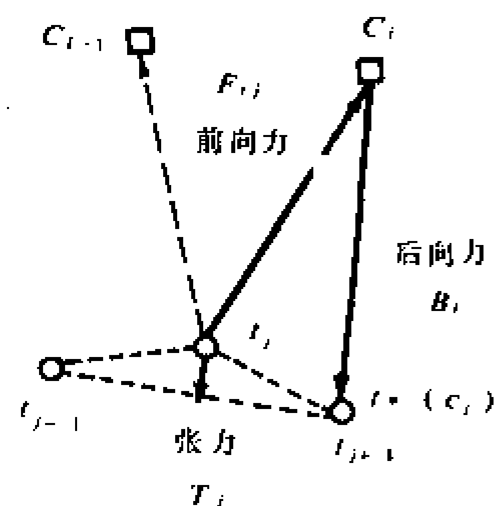


图 14.6 运动点 t_j 和城市 C_i 间的受力图

综上所述，弹性网算法就是同时求解一组差分方程的过程，这是一个并行操作过程，适合于硬件上的

并行实现．具体模拟时，还需确定 α、β，路径中的点的个数（如 $2N$）和参数 K 的初值和递减规律，详见文[4]。

用弹性网法所得到的解，其质量是很高的．对 100 个城市所得到的次优解仅比最优解差 1%，但需要相当长的收敛时间．Burr[2]利用对称原理进行的改进工作大大减少了所需的计算时间．如前所述，Durbin—Willshaw 只利用了两种力，即城市对运动点的前向拉力 F_{ij}和顶点间的张力 T_j，如图 14.6 所示。

Burr 引入第三种后向力 B_i，它由距城市 C_i 最近顶点 $t^*(C_i)$所决定，可看成是把城市拉向周游路径．三种力的合成效果使得运动点向城市分布较密集的区域运动，而周游路径最短由张力 T_j来保证．运动点 t_j的运动方程是：

$$\Delta t_j = \alpha\sum_i (W_{ij}F_{ij} - S_{ij}B_i) + T_j$$

$$= \alpha\sum_i \left[(W_{ij}(C_i - t_j) + S_{ij}(C_i - t^*(C_i)) \right] + (t_{j-1} + t_{j+1} - 2t_j)/2$$

其中 W_{ij} 如前所述，$S_{ij} = \varphi(\ |t^*(C_i) - t_j|, k) / \sum_k \varphi(\ |t^*(C_i) - t_k|, k)$；$t^*(C_i)$ 是一函数，它给出与城市 C_i 最近的点的位置．

若把上式中的第一项看作是确定距 C_i 最近点的期望值，那么引入的第二项则是在计算真正的最近点．模糊计算和精确计算相结合，可使每步的决定更加准确，从而提高了收敛速度．

为保证解的高质量，平滑参数 K 取平均接近距离，即

$$K = \beta\left[\sum_i |C_i - t^*(C_i)|^2 / N \right]^{\frac{1}{2}}$$

这里的 K 相当于第三小节中模拟退火时温度的作用，它逐渐减少导致了解的质量不断提高．

值得指出的是：多个 TSP 问题[19]在弹性网模型中很容易解决．只要放置多个初始圆让它们按照上述原则不断扩张，最终将得到多个 TSP 问题的解答．不过初始圆的位置将与最终解有关，见图 14.7。

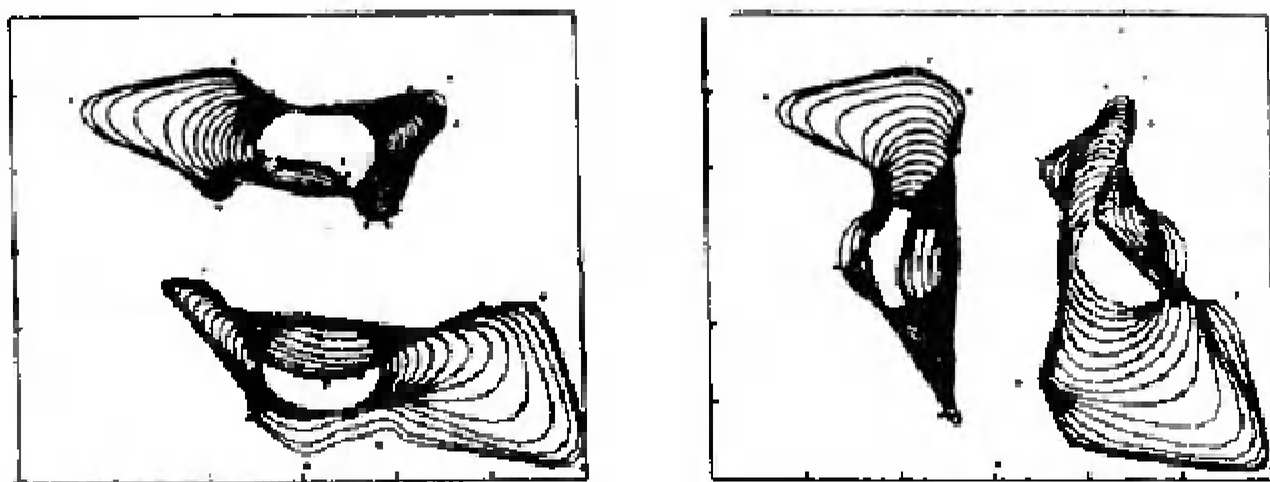


图 14.7　多张弹性网下的多 TSP 问题

14.3.2　自组织映射

现实世界中神经元间的联系并不是一成不变的，而是一个随外部环境不断演化的过

程。Kohenen[11]所提出的自组织映射(Self-Organizing Maps)较好地模拟了神经元在无导师状态下适应环境变化而自学习的过程，具有理论和实际应用上的重要意义。本小节着重探讨自组织映射的计算能力，以 TSP 问题为例说明自组织映射在组合优化中的应用。

1. 模型简介

Kohenen 自组织映射是一个拓扑保序映射，它把高维空间 V 的信号映射到低维空间 A（通常是二维离散空间）并尽可能地保持拓扑（相邻）关系不变。如图 14.8 所示，手掌面的五个区域经映射后虽然位置已有所改变，但相邻关系保持不变。如 T 和 D、L、M、R 都还相邻等。Kohenen 自组织模型正是基于这种拓扑保序映射，它通过 A 空间上的自学习算法，体现了高维空间 V 的某些重要特征，以适应外部的环境。

一般的 Kohenen 自组织映射学习算法可描述如下：

设 $v\in V$ 是高维空间中的一个输入信号，在每一学习步骤中，v 依概率 $P(v)$而定。二维网格 A 中 r 处有一向量 $W_r\in V$，它把 A 中的地址 r 映射到 V 的点 W_r，并以如下规律变化：

(1) 确定最近点 s，使得 $\|W_s-v\|=\min\limits_{r\in A}(\|W_r-v\|)$。其中 v 是当前输入的随机向量。

(2) 对所有 s 的邻居 r，修改 W_r，即 $\Delta W_r=\varepsilon h_{rs}(v-W_r)$。这里 h_{rs} 是一预先指定的邻居修改函数，通常 $0\leqslant h_{rs}\leqslant 1$，且在 $r=s$ 处最大，并随着 $\|r-s\|$ 的增大而趋于零，即仅在 r 的邻居范围内起作用。而 ε 是一个反映学习步长大小的参数。这样，随着学习的不断深入，ε 和 h_{rs} 的作用越来越小，上述算法最终产生的 r 处向量 W_r 将和 V 中的某些顶点对应并保持拓扑的不变性。

在下面，我们将讨论如何选择 h_{rs} 和 ε 以解决 TSP 这一图论中的难题。

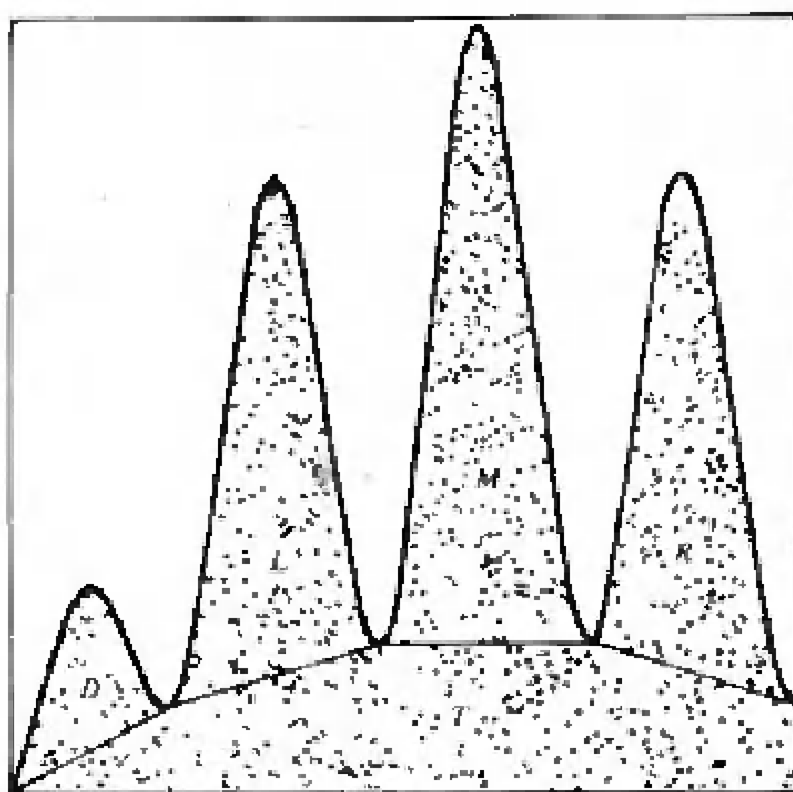


图 14.8 自组织映射的拓扑保序性

2. 自组织映射下的 TSP 问题

用 Kohenen 模型求解 TSP 问题的直观想法和弹性网类似[5]：用圆周点到城市点的拓

扑保序映射，使得最终周游路径上点的分布概率和城市点一致，从而得到一个较好的周游路径。

形式化的算法描述如下：

设 A_1，A_2，…，A_n 是 $[0,1]^2$ 上的城市点．x_0，x_1，…，x_{N-1} 是 $[0,1]^2$ 上的一个环．设 $l_0 \ll N$，x_i 的邻居 $r(i)$ 定义为：$r(i)=\{j|j\in[i-l_0, i+l_0] \bmod N\}$。环中各点在 t 时刻的位置由 $X^t=(x_o^t, x_1^t, \cdots, X_{N-1}^t)$ 表示，初始时：$X^0=(x_0, x_1, \cdots, x_{N-1})$．

令 ζ^1，ζ^2，…，ζ^t，… 是一系列在 A'_i 上均匀分布的独立随机变量，即 ξ^t 在 t 时刻取某个城市 A_i 的概率是 $1/n$．那么 TSP 算法是：

(1) 确定下标 $i_0(t)$，使得

$$\|\xi^{t+1}x^t_{i_0(t)}\|=\min_i\left\{\|\xi^{t+1}x_i^t\|\right\}$$

即与 ξ^{t+1} 最近的点 $x^t_{i_0(t)}$.

(2) 修改规则如下：

$$x_i^{t+1}=(1-\varepsilon)x_i^t+\varepsilon\xi^{t+1}, \qquad i\in r(i_0(t))$$

$$x_i^{t+1}=x_i^t, \qquad \text{其它}$$

其中，$t\to\infty$，$\varepsilon\to 0$.

再指定 l_o（邻居）、ε（步长）的变化规律[5]就可用 Kohenen 模型对 TSP 问题求解了．

上述算法中，距离的作用并不突出，而 TSP 问题所关心的恰是最短周游路径．为此，下面我们经过一些简单的运算，得到显示的距离表示形式．

设 $A_{jk}=\|A_jA_k\|^2$，城市 j 和点 k 的距离平方，

$D^t_{ij}=\|x_i^tA_j\|^2$，t 时刻两点 i 和 j 的距离平方，

则矢量A、B、C组成一个三角形．根据三角等式，我们有：

$$\|AB\|^2+\|AC\|^2-2AB\cdot AC=\|BC\|^2$$

设 $\xi^{t+1}=A_{j_o}$，那么对所有 j_o 的邻居 $i\in r(j_o(t))$

$$D^{t+1}_{ij}-D^t_{ij}=\|A_jx_i^{t+1}-A_{j_o}x_i^{t+1}+A_{j_o}x_i^{t+1}\|^2-\|A_jx_i^t\|^2$$

$$=\|A_jx_i^{t+1}\|^2+2A_jA_{j_o}\cdot A_{j_o}x_i^{t+1}+\|A_{j_o}x_i^{t+1}\|^2-\|A_jx_i^t\|^2$$

代入 x_i^{t+1}，则

$$D^{t+1}_{ij}-D^t_{ij}=\|A_jA_{j_o}\|^2+2(1-\varepsilon)A_jA_{j_o}\cdot A_{j_o}x_i^t+(1-\varepsilon)^2\|A_{j_o}x_i^t\|^2-\|A_jx_i^t\|^2$$

利用三角等式，可得：

$$D^{t+1}_{ij}-D^t_{ij}=\Delta_{jj_o}-(1-\varepsilon)\left[\Delta_{jj_o}+D^t_{ij_o}-D^t_{ij}\right]+(1-\varepsilon)^2D_{ij_o}-D_{ij}$$

因此，修改规则如下：

$$\begin{cases} D_{ij}^{t+1} = D_{ij}^{t} + \varepsilon\left[\Delta_{ij_o} - \left(D_{ij_o}^{t} + D_{ij}^{t} \right) + \varepsilon D_{ij_o}^{t} \right], & i \in r(j_o(t)) \\ D_{ij}^{t+1} = D_{ij}^{t}, & \text{其它} \end{cases}$$

这里 D_{ij} 就可看成是神经元间的连接强度。Kohenen 学习算法使得某些神经元的连接强度增强，某些逐渐减弱，运动顶点不断向城市靠扰而形成一条周游路径，见图 14.9 所示。

模拟当中一般取 $\varepsilon = 1/(1+\log(t+1))$ 或 $\varepsilon = \delta r^t \varepsilon_o$ $(0<\delta<1)$，$N=2n \sim 2.5n$，详细情况及结果分析见文[5]。

最后需要说明的是：尽管弹性网和自组织映射有相似之处但它们存在着如下区别。首先，弹性网法是确定性算法而自组织映射是概率型的；其次自组织映射比较通用而弹性网中的能量函数是为 TSP 特制的。

14.3.3 模拟退火

模拟退火(Simulated Annealing，以下简称 SA)算法的思想最早是由 N.Metropolis 等人于 1953 年提出的，但把它用于组合优化却是在 1983 年由 S.Kivkpatrick 等人[10]给出的。模拟退火算法将组合优化问题与统计力学中的热平衡问题类比，另辟了一条求解组合优化问题的新途径。它通过模拟退火过程，可找到全局（或近似）最优解。模拟退火算法是基于 Monte Carlo 迭代求解法的一种启发式随机搜索算法。SA 算法用于解决组合优化问题的出发点是基于物理中晶体物质的退火过程与一般组合问题间的相似性。在对固体物质进行退火处理时，通常先将它加温熔化，使其中的粒子可自由运动，然后随着温度的逐渐下降，粒子也逐渐形成了低能态的晶格。若在凝结点附近的温度下降速率足够慢，则固体物质一定会形成最低能量的基态。对于组合优化问题来说，它也有这样的类似过程。组合优化问题解空间中的每一点都代表一个具有不同目标函数值的解。所谓优化，就是在解空间中寻找目标函数最小（大）解的过程。若把目标函数看成能量函数，某一控制参数视为温度 T，解空间当作状态空间，那么 SA 算法寻找基态的过程也就是求目标函数极小值的优化过程。

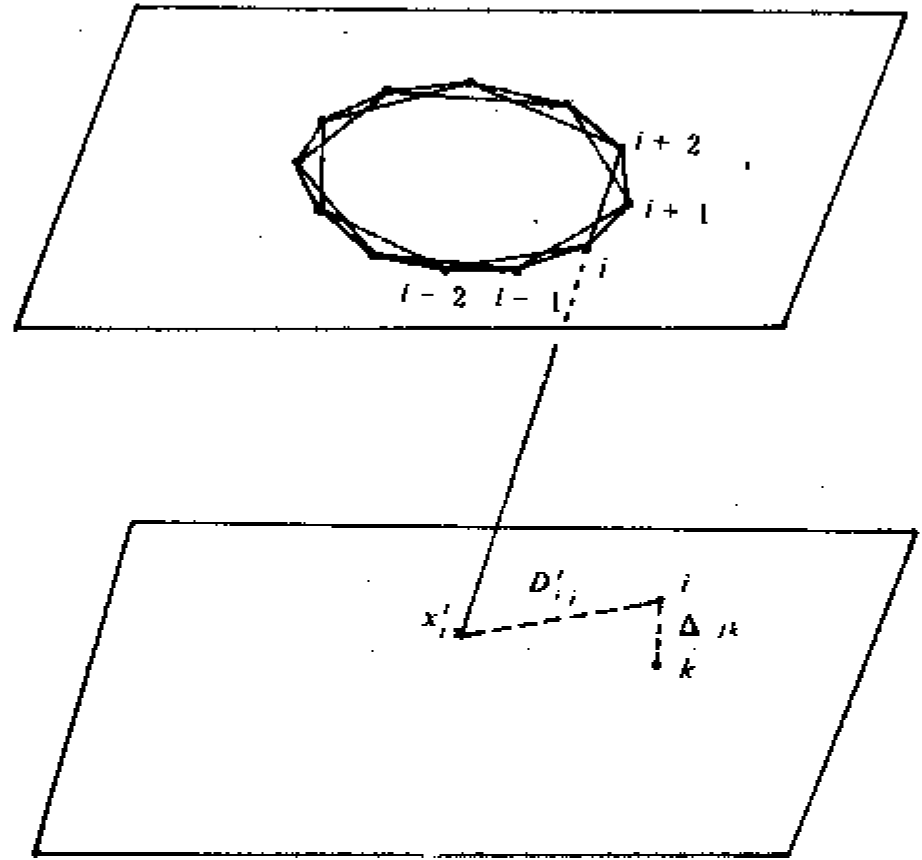


图 14.9 双层网络：上层是环，下层是城市

1. 模拟退火简介

设 $S=\{s_1, \cdots, s_k\}$ 为所有可能的组合状态集合，$C: S \rightarrow R$ 为非负目标函数，即 $C(s_i) \geqslant 0$ 反映取状态 s_i 为解的代价，则组合优化问题可形式地表述为寻找 $s^* \in S$，使得

$$C(s^*) = \min C(s_i) \qquad \forall\, s_i \in S$$

SA 法的基本思想是：把每种组合状态 s_i 看成某一物质体系的微观状态，而 $C(s_i)$ 看成该物质体系在状态 s_i 下的能量，并用控制参数 T 表示伪温度．让 T 从一个足够高的值慢慢下降，对每个 T，用 Metropolis 抽样法模拟该体系在此 T 下的热平衡态。即对当前状态 s 做随机扰动产生一个新状态 s'，计算增量 $\Delta C' = C(s') - C(s)$，并以概率 $\exp(-\Delta C / bT)$ 接受 s' 作为新的当前状态．当这样的随机扰动重复足够次数后，系统将达到该温度下的热平衡态，并且系统的状态将导致 Boltzmann 分布，b 是 Boltzmann 常数．

模拟退火算法的主要步骤描述如下：

算法 14.1　SA-ALGORITHM

```
procedure  SA__Algorithm(i0,T0);
/ * s_{i0} 为任一初态，T0 为初始控制参数 * /
begin
(1)  s←s_{i0} ;     C*←C_i ;     k←0
     / * 简记 C_i = C(s_i)，C* 为当前代价值． * /
(2)  repeat
   (2.1)  repeat
      (2.1.1)  s_j ←Generate(s);
      (2.1.2)  if C_j ≤ C* then s←s_j
                         else if Accept (j, s)  then s←s_j    endif
               endif
      (2.1.3) until  内循环结束;
   (2.2)  T_{k+1}←Update(T_k);  k←k+1
(3)  until  T_k ≤ ε
end.
```

上述算法中，(1)为初始化．(2.1.1)的 Generate(s)表示从 s 的邻域中随机产生下一个状态 s_{j_0} ．若 $C_j \leqslant C^*$，则接受 j 为新的当前状态；否则仅以一定的概率接受 j 为新的当前状态，这也就是 Accept(j,s)函数的功能．Accept 函数通常取如下形式：

```
function  Accept (j ,s);
   begin
      ΔC'←C_j −C* ;
      if exp(−ΔC' / bT_k) > Random(0,1)   / * b 为 Boltzmann 常数 * /
         then  Accept←true
         else  Accept←false
      endif
   end.
```

(2.1.3)中的内循环结束条件是指在每一温度 T_k 下迭代多少次以达到平衡态，而(2.2)中的 Update(T_k)函数则表示温度每次下降的速率。由此可知，SA 算法有 3 个重要函数：产生函数 Generate，接受函数 Accept 和温度更新函数 Update。在实际应用中，初始温度 T_0，内循环次数和终止条件也是影响 SA 算法性能的重要参数。下面结合 TSP 问题，讨论如何确定函数形式和各参数。

2. 用模拟退火法求解 TSP 问题

传统的启发式算法主要有两种：第一是先自顶向下的分而治之、再自底向上的组合求解的分治法。第二是步步贪心以达到全局最优的迭代法。模拟退火法可看成是上述两种方法的综合。在高温下，它是大粒度的分治法；而在低温下是细粒度的贪心法。由于 SA 算法中能做概率性的扰动以跳出局部极小，所以它得到的解的质量一般很高。

在 TSP 问题中，设 $\sigma=(i_0,i_1,\cdots,i_N)$为顶点的一个排列，则优化目标函数就可取为路径长度，即：$c(s_i)=\sum_j \|A_jA_{\sigma_i(j)}\|$，$j\in[1,N]$。产生函数 Generate 可定义为：

```
function   TSP__Generate(s_i);
/ * 令 s_i=(i_1,…,i_N) * /
begin
    i←Random(1,N−1) ;  j←Random(i+1,N) ;
    return (Swap(i,  j,  s_i,  s_j ))
end.
```

它表示：交换顶点 A_i 和 A_j 看看能否产生较短的周游路径，即可用弧(X_{i-1},X_j)、(X_j,X_{i+1})、(X_{j-1},X_i)、(X_i,X_{j+1})去替代弧(X_{i-1}, X_i)、(X_i, X_{i+1})、(X_{j-1}, X_j)和(X_j, X_{j+1})。

温度更新函数 Update(T)可取：$T_{k+1}=d/(1+\log(1+k))$它在一定条件下能保证 SA 算法收敛到最佳解，但计算时间太长，故在实际应用中通常取：$T_{k+1}=\lambda T_k$，$0<\lambda<1$。

检验 Metropolis 抽样法是否稳定，一般取：

① 检验$C(s_i)$的均值是否稳定；

② $C(s_i)$变化很小。

总之，对 TSP 问题而言，模拟退火算法是诸神经网模型中最好的一种（指解的质量），目前所能处理的城市数目已达 6000 之多。

14.3.4 均场退火

从上一小节介绍中可以看出，模拟退火算法是一个通用的、与领域无关的、具有概率爬山性的、强有力的组合优化算法。但它要求温度下降得足够慢，导致了该算法的计算时间很长。均场退火(Average Field Annealing)即可看作是一种新型的神经网计算模型；又可视为是对模拟退火的重大改进。它只需要在某个关键温度附近实施退火过程就可取得较

好的效果，使得该算法的时间性能有很大提高.

1. 模型简介

均场退火可由下述方程刻化:

$$E = f(V) \tag{14.3.1}$$

$$E_i = \frac{\partial E}{\partial v_i} = \frac{\partial f}{\partial v_i} \tag{14.3.2}$$

$$v_i = g(E_i, T) \tag{14.3.3}$$

其中: $V=(v_1,v_2,\cdots,v_N)$是能量函数中 N 个相互制约的组合优化变量，E_i 称为均场(Mean Field)，g 是神经元输入输出的 Sigmoid 函数，T 是温度控制参数，它与 Hopfield 模型中的增益系数的倒数或模拟退火中的温度作用相当.

所谓均场退火就是在某一个关键温度 T_c 附近，按照上述模型实施模拟退火过程，以便达到某个热平衡态，这时得到系统的一个较优解. 下面结合 TSP 问题的讨论，介绍用均场退火解决组合优化问题的一般方法.

2. 均场退火下的 TSP 问题求解

针对 Hopfield 模型中网络参数的难确定性和解的质量不高等缺点，Van den Bout 等人[18]提出了如下改进方案:

(1) 能量函数只有两项，即

$$E = \frac{d_p}{2}\sum_i\sum_X\sum_{Y\neq X} V_{Xi}V_{Yi} + \frac{1}{2}\sum_X\sum_{Y\neq X}\sum_i d_{XY}V_{Xi}(V_{Y,i+1}+V_{Y,i-1}) \tag{14.3.4}$$

它只保留了原 Hopfield 模型中置换矩阵表示 TSP 问题的列约束，这样就很容易确定参数 d_p，即 $d_p = \max_{i,j}\{2d_{ij}\}+\varepsilon$. 由此可知: d_p 为两点间最大距离的 2 倍多一点，用以保证形成合法解. 其他约束条件由下面的规一化过程所满足.

(2) 神经元规一化

神经元的输出状态变量 V_{Xi} 看作是当城市在作随机热平衡扰动中城市 X 被第 i 次访问的概率，它遵从 Boltzmann 分布，即 $V_{X_i} \approx \exp(-E_{Xi}/T)$ 其中 E_{Xi} 是均场，可由 (14.3.2) 中算出. 即

$$E_{Xi} = d_p\sum_{Y\neq X}V_{Yi} + \sum_{Y\neq X}d_{XY}(V_{Y,i+1}+V_{Y,i-1}) \tag{14.3.5}$$

这样均场值 E_{Xi} 大的，其位置占有概率 V_{Xi} 小，说明某城市 X 不太可能在周游路径中第 i 次被访问. T 越小越容易得到合法解，T 越大解的质量越高. 为了得到真正的概率，对神经元的输出 V_{Xi} 进行归一化:

$$V_{Xi} = \exp(-E_{Xi}/T)/[\sum_j\exp(-E_{Xj}/T)] \tag{14.3.6}$$

它保证了每个城市只在路径中出现一次($\sum_i V_{Xi}=1$)，起到了原置换矩阵表示中行的约束作用.

从(14.3.5)和(14.3.6)两式中可知：在高温时，城市被访问的位置占有概率是均匀分布的：

$$T \to \infty \Rightarrow V_{Xi} \to 1/n \qquad \forall X, i$$

在低温时，城市向具有较小均值的位置凝结，它使得整个周游路径最短。假如两个城市占据了相同的位置，必然会留下某个空位置，这种情形是维持不住的。因为在 d_p 大于任意两点间最大距离两倍时，两个城市无论哪个占据空位置都将得到较小的能量，所以系统将向这种状态演变，见图 14.10。这样，通过目标函数中列约束条件和归一化的联合作用将保证产生合法解。算法过程如下：

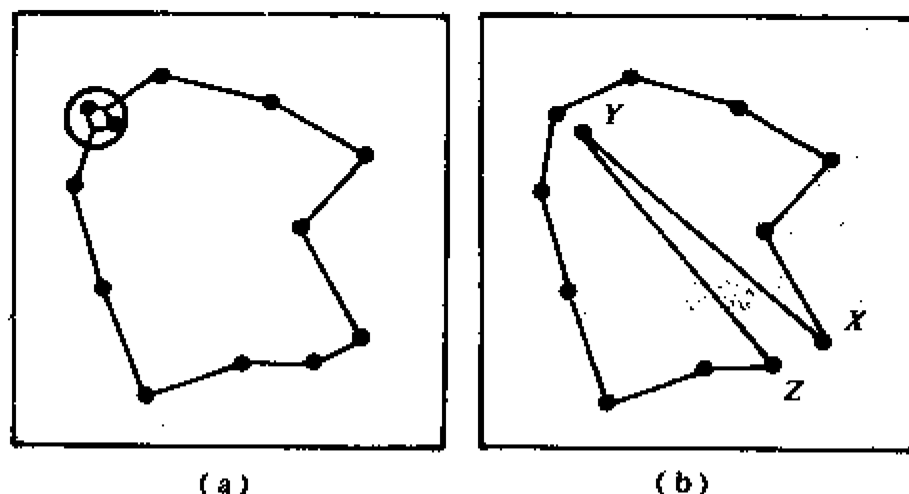


图 14.10 (a) 非法路径由于两个城市占据了同一位置

(b) 合法路径由于 d_p 的作用：$d_{XY}+d_{YZ}<d_p$

算法 14.2 AFA-ALGORITHM

procedure AFA;

begin

$T \leftarrow T_s$;

repeat

Select a city X at random; Sum $\leftarrow 0$;

for $i \leftarrow 0$ **to** $n-1$ **do**

$E_{Xi} \leftarrow d_p \sum_{Y \neq X} V_{Yi} + \sum_{Y \neq X} d_{XY}(V_{Y,i+1} + V_{Y,i-1})$;

Sum $\leftarrow$ Sum $+ \exp(-E_{Xi}/T)$

endfor;

for $i \leftarrow 0$ **to** $n-1$ **do**

$V_{Xi} \leftarrow \exp(-E_{Xi}/T)/\text{Sum}$

endfor;

$E \leftarrow \frac{d_p}{2}\sum_i \sum_X \sum_{Y \neq X} V_{Xi} V_{Yi} + \frac{1}{2}\sum_X \sum_Y \sum_i d_{XY} V_{Xi}(V_{Y,i+1} + V_{Y,i-1}))$

until $(\Delta E = 0)$

end.

其中 $\Delta E=0$ 是热平衡条件，T_s 是某一温度。均场退火的关键是在某一关键温度 T_c 附近进行模拟退火，以节省时间，得到较好的解。

3. 关键温度 T_c

图 4.11 是能量函数 E 随温度 T 的变化情况，从图中可见，在关键温度 T_c 以上进行退火对解没有多大影响，只是起到了搜索

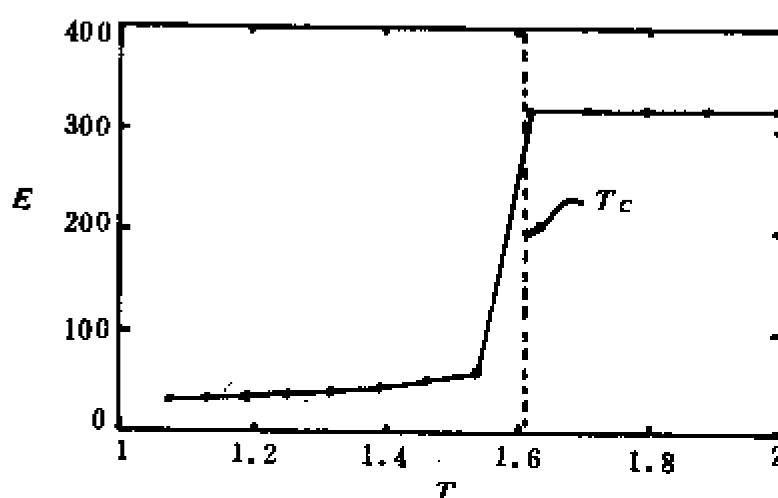


图 14.11 能量函数 E 和温度 T 的变化关系

T_c 的作用．在低温时，退火对解也不起多大作用，充其量是使输出趋向于 0 或 1 而已。均场退火只是在关键温度 T_c 附近有效果．它提示我们：一旦找到了准确的 T_c，只需在其附近进行退火过程即可．下面介绍一种 T_c 的估算方法。

从图 14.11 得知：在关键温度 T_c 附近能量发生巨变，即一个微小的扰动将在整个系统中扩散。可以通过对均场(14.3.5)的研究来考察这个温度下的传播效应。

设某城市 Y 被第 i+1 次访问的占有概率有一个微扰 $\Delta V_{Y,i+1}$，它将对城市 X 被第 i 次访问的均场有如下影响：

$$\Delta E^1_{Xi} = d_{XY}\Delta V_{Y,i+1} \tag{14.3.7}$$

因为在 (14.3.5) 中只有第二项含有 $V_{Y,i+1}$，它对 (14.3.7) 有贡献（同理，$\Delta E^1_{x,i+2} = \Delta E^1_{Xi}$）。利用 (14.3.6) 的可导性，可得均场变化对周游路径上位置 i 的占有概率作用是：

$$\begin{aligned}\Delta V_{Xi} &= \Delta E^1_{Xi} \cdot \mathrm{d}V_{Xi} / \mathrm{d}E_{Xi} \\ &= (d_{XY}\Delta V_{Y,i+1}) \cdot (\exp(-E_{Xi}/T) / (\sum_j \exp(-E_{xj}/T)))' \\ &= d_{XY}\Delta V_{Y,i+1} \cdot V_{Xi}(V_{Xi}-1)/T \end{aligned}\tag{14.3.8}$$

这里我们忽略了 $\Delta E_{X,i+2}$ 对 V_{Xi} 的微弱影响。从(14.3.8)式可知：城市 Y 在 i+1 处的占有概率的增加($\Delta V_{Y,i+1}>0$)将导致其他城市在 i 处的占有概率的减少($\Delta V_{Xi}<0$)。对这一结论可以这样理解：(14.3.8)式来自于周游路径最短这一优化目标项，$\Delta V_{Y,i+1}$ 的增加使能量函数的第二项有所增加．欲使路径最短，必然某些东西要减少，以便抑制周游路径的增长．因此其他城市的占有概率自然要减小。与此同时，我们不要忘记对这一斥力的反作用，即其他城市在 i 位置上形成的部分真空将导致约束条件不满足，它将在 i 处产生对某个城市 X 的引力。从(14.3.5)式中第一项可知，这一引力对 X 的均场作用是：

$$\Delta E^2_{Xi} = \sum_{Z\neq X}\frac{\partial E_{Xi}}{\partial V_{Zi}}\Delta V_{Zi} = d_p\sum_{Z\neq X}\Delta V_{Zi} \tag{14.3.9}$$

综合(14.3.7)和(14.3.9)，可得总变化是：

$$\begin{aligned}\Delta E_{Xi} &= \Delta E^1_{Xi} + \Delta E^2_{Xi} \\ &= d_{XY}\Delta V_{Y,i+1} + d_p\sum_{Z\neq X}\Delta V_{Zi}\end{aligned}$$

占有概率的微扰是：

$$\begin{aligned}\Delta V_{Xi} &= \Delta E_{Xi} \cdot V_{Xi}(V_{Xi}-1)/T \\ &= \{d_{XY}\Delta V_{Y,i+1} + d_p\sum_{Z\neq X}\Delta V_{Zi}\}V_{Xi}(V_{Xi}-1)/T\end{aligned}$$

用(14.3.8)中的ΔV_{Xi}近似取代ΔV_{Zi}，可得：

$$\Delta V_{Xi} = \{d_{XY} + d_p\sum_{Z\neq X}d_{ZY}V_{Zi}(V_{Zi}-1)/T_X\}\Delta V_{Y,i+1} \cdot V_{Xi}(V_{Xi}-1)/T \tag{14.3.10}$$

每个城市有着它自己的临界温度 T_X，在这个温度之下，引力的作用将大于斥力，周游路径开始形成。这个温度可估算如下：

$$\Delta V_{Xi}|_{T=T_X}=0$$

即

$$(d_{XY}+d_p\sum_{Z\neq X}d_{ZY}V_{Zi}(V_{Zi}-1)/T_X)\Delta V_{Y,i+1}\cdot V_{Xi}(V_{Xi}-1)/T=0$$

求解得

$$d_{XY}=\sum_{Z\neq X}d_p d_{ZY}V_{Zi}(1-V_{Zi})/T_X$$

我们欲求最高的临界温度 T_X，这时可认为在高温下，每个城市对各位置的占有概率是 $1/N$，代人上式得:

$$d_{XY}=d_p\sum_{Z\neq X}d_{ZY}(\frac{1}{N})(1-\frac{1}{N})/T_X$$

令 $\sum_{Z\neq X}d_{ZY}=(N-1)\bar{d}$，$\bar{d}$ 是平均两点距离，那么 $d_{XY}=d_p\bar{d}(1-\frac{1}{N})^2/T_X$，当 N 很大时，可近似求得:

$$T_X\approx d_p\bar{d}/d_{XY} \tag{14.3.11}$$

我们估算出了每个城市的临界温度，这样只要取 $T_c=\max_X\{T_X\}$ 作为整个系统的临界温度即可．但为了更准确些，还需进一步推导．

设 $\Delta V_{Y,i+1}$ 对 ΔV_{Xi} 和 $\Delta V_{X,i+2}$ 施加大体相同的作用以保证自身的稳定性，即 $\Delta V_{Xi}=\Delta V_{X,i+2}=\Delta V_{Y,i+1}/2$，可得:

$$\Delta V_{Y,i+1}/2=\{d_{XY}+d_p\sum_{Z\neq X}d_{ZY}V_{Zi}(V_{Zi}-1)/T_c\}\cdot\Delta V_{Y,i+1}V_{Xi}(V_{Xi}-1)/T_c$$

利用和从前同样的假定，可得:

$$1/NT_c(d_p\bar{d}/T_c-d_{XY})\approx 1/2$$

T_c 应该在 d_{XY} 小的地方出现，当 $d_{XY}\approx 0$ 时，

$$T_c=\sqrt{\frac{2d_p\bar{d}}{N}} \tag{14.3.12}$$

这样我们就可以在 T_c 附近进行模拟退火，用数次热平衡态模拟来代替整个的缓慢退火过程，从而节省时间，得到较优解．

14.4 应用举例

由于 Hopfield 模型既具有丰富的动力学行为；又简单实用便于 VLSI 的实现，本节将以它为基础讨论神经计算在图论中的应用[14]．用神经网模型解决组合优化问题需涉及到模型选择，问题表示，能量函数，运动方程和网络参数等几个关键步骤．在前面章节中，已对图论中的旅行商问题作了详细介绍，本节只采用 Hopfield 模型来解决问题，因此我们将从问题描述，问题表示，能量函数、运动方程和网格参数这几个方面来说明用神

经网模型解决图论问题的基本方法．值得指出的是：问题表示和能量函数的形式并不是唯一的。我们所给出的只是一种较好的形式，如何设计出更好的表达方式和相应的能量函数是神经优化计算目前研究中的热点．

1. 图的二划分

1) 问题的描述

令 $G(V,E)$ 是一个无向图，$|V|=n$，$|E|=m$，顶点 V_i 有权 $W_i \geqslant 0$，边 (i,j) 有权 $e_{ij} \geqslant 0$，G 的二划分是指将 G 的顶点划分成两个不相交的子集 S_1 和 S_2 ($S_1 \cup S_2 = V, S_1 \cap S_2 = \phi$)，在两个顶点子集中所含顶点权和 $\sum_{j\in S_i} W_j (i=1$ 或 $2)$ 大致相等的情况下使得 S_1 和 S_2 间的边权和最小，即 $\min \sum_{i\in S_1}\sum_{j\in S_2} e_{ij}$.

2) 问题表示

可用神经元的状态来表示它对划分集合的归属，即 $V_i=1$ 表示该神经元所代表的顶点属于 S_1；$V_i=0$ 表示它属于 S_2．总共需要 n 个神经元。

3) 能量函数

$$E=\frac{A}{2}\sum_i\sum_j e_{ij}(V_i\overline{V}_j+\overline{V}_iV_j)+B\sum_i\sum_j W_iW_j(V_iV_j+\overline{V}_i\overline{V}_j) \tag{14.4.1}$$

其中：$\overline{V}_i=1-V_i$．第一项 $\sum_{i\in S_1}\sum_{j\in S_2} e_{ij}$ 为优化目标项．因为当 $V_i=1$，即顶点 $i\in S_1$ 时，要使乘积项 $V_iV_j\neq 0$，则 $\overline{V}_j=1$，即 $V_j=0$，$V_j\in S_2$．第二项表示的是 $(\sum_{j\in S_1} w_j)^2 + (\sum_{j\in S_2} w_j)^2$．因 $a>0$，$b>0$ 时，有 $a^2+b^2\geqslant 2ab$。所以，当且仅当 $a=b$ 时第二项取最小值．即第二项描述的是当且仅当划分后顶点权和大致相等时取极小．所以能量函数 (14.4.1) 式刻划了图的二划分中的优化问题．

4) 运动方程

第 i 个神经元的运动方程是：

$$\frac{\mathrm{d}u_i}{\mathrm{d}t}=-\frac{u_i}{\tau}-\frac{\partial E}{\partial V_i}$$

代入(14.4.1)式可得：

$$\frac{\mathrm{d}u_i}{\mathrm{d}t}=-\frac{u_i}{\tau}-A\sum_i e_{ij}(1-2V_j)-2B\sum_j w_iw_j(2V_j-1) \tag{14.4.2}$$

5) 网络参数

在Hopfield连续模型中，偏流 I_i 和矩阵元素 T_{ij} 可由下式求出：

$$I_i=\text{Constant_of}\left(\frac{\partial E}{\partial V_i}\right)$$

$$T_{ij} = \text{Constant_of}\left(\frac{\partial E}{\partial V_i \partial V_j}\right)$$

其中 Constant_of( ) 是一函数，从它可得到多项式的常数项．这样，对二划分问题来说，

$$I_i = A\sum_j e_{ij} - 2Bw_i\sum_j w_j \tag{14.4.3}$$

$$T_{ij} = -2Ae_{ij} + 4Bw_iw_j \tag{14.4.4}$$

2. 图的K划分

1) 问题的描述

图 $G(V,E)$ 的 K 划分是指将 G 的顶点划分成 K 个彼此不相交的子集 S_i，即 $\bigcup_i^K S_i = V$，$S_i \cap S_j = \Phi$，$i \neq j, i,j = 1,\cdots,K$，在保持各划分内顶点权和大致相等的情况下，使得任意两划分间的边权和尽可能的小．

2) 问题表示

对于图的 $K(K>2)$ 划分问题，就不能简单地用神经元的最终二值状态来表示了，最好用置换矩阵表示．矩阵的行代表顶点，列代表所属的划分集合．神经元 (i,j) 的输出 $V_{ij}=1$ 则表示图的第 i 个顶点应属于第 j 个划分．总共需要 $N\times K$ 个神经元．

3) 能量函数

$$E = \frac{A}{2}\sum_X^K\sum_{Y\neq X}^K\sum_i^n V_{iX}V_{iY} + \frac{B}{2}\left(\sum_i^n\sum_X^K V_{iX} - n\right)^2 + \frac{C}{2}\sum_i^n\sum_j^n\sum_X^K w_iw_jV_{iX}V_{jX}$$
$$+\frac{D}{2}\sum_i^n\sum_j^n\sum_X^K e_{ij}V_{iX}V_{jY} \tag{14.4.5}$$

E 中的第四项是优化目标项，它最小化各划分间的边权和．第一项是行约束，表示每个顶点最多属于一个划分集合．第二项是全局约束，它要求有 n 个 1 使得全部顶点被划分．第三项表示：$C\sum_X^K(\sum_{i\in S_X} w_i)^2$．它同样利用算术平均和几何平均的关系表示了各划分内顶点权和大致相等时取极小的约束条件．

4) 运动方程

$$\frac{\mathrm{d}u_{iX}}{\mathrm{d}t} = -\frac{u_{iX}}{\tau} - A\sum_{Y\neq X}^K V_{iY} - B\left(\sum_j^n\sum_Y^K V_{jY} - n\right) - C\sum_j^n w_iw_jV_{jX}$$
$$-D\sum_j^n\sum_{Y\neq X}^K e_{ij}V_{jY} \tag{14.4.6}$$

它表示第 (i,X) 个神经元的运动方程，为不混淆起见，我们对部分项进行了下标替换．

5) 网络参数

$$I_{iX} = Bn \tag{14.4.7}$$

$$T_{iX,jY} = -A\delta_{ij}(1-\delta_{XY}) - B - Cw_iw_j\delta_{XY} - De_{ij}(1-\delta_{XY}) \tag{14.4.8}$$

这里简单介绍一下网络参数中 δ 函数是如何得出的．从 (14.4.6) 式的第二项 $A\sum_{Y\neq X}^{K} V_{i}=E_{1}$ 中可知（为讨论方便，下标进行了替换），$\frac{\partial E_{1}}{\partial V_{jY}}$ 仅当 $i=j$ 和 $Z\neq X$ 时有效 (即 $\neq 0$)，因此网络参数中含有 $\delta_{ij}(1-\delta_{XY})$．即从直观上看，如果求和中有 $X\neq Y$ 这一条件，那么连接矩阵中必有 $(1-\delta_{XY})$ 这一项．

3. 图的顶点覆盖

1) 问题描述

图 $G(V,E)$ 的顶点覆盖是 G 的一个子集 $S_{1}\subseteq V$，使得 G 的每条边至少有一个顶点在 S_{1} 中．图的顶点覆盖问题旨在找出具有最少元素个数的 S_{1}．

2) 问题表示

与图的二划分问题类似，可用神经元的二值输出状态表示该神经元的归属．当网络稳定时，凡是 $V_{i}=1$ 的顶点属于最小覆盖集；而 $V_{i}=0$ 的那些顶点则不属于最小覆盖集．整个问题可用 n 个神经元表示．

3) 能量函数

假定图 G 用邻接矩阵 $A=(a_{ij})_{n\times n}$ 表示，则

$$E=\frac{A}{2}\sum_{i}\sum_{j}V_{i}V_{j}+\frac{B}{2}\sum_{i}\sum_{j}a_{ij}\overline{V}_{i}\overline{V}_{j} \tag{14.4.9}$$

其中 $\overline{V}_{i}=1-V_{i}$．第一项是优化项，它表示 $(\sum_{i\in S_{1}}V_{i})^{2}$，即覆盖集大小的平方．第二项是约束项，只要 V_{i} 或 V_{j} 中的一个属于 S_{1}，相应的 $\overline{V}_{i}$ 或 $\overline{V}_{j}$ 为 0，该项取最小值．

4) 运动方程

$$\frac{\mathrm{d}u_{i}}{\mathrm{d}t}=-\frac{u_{i}}{\tau}-A\sum_{j}V_{j}+B\sum_{j}a_{ij}(1-V_{j}) \tag{14.4.10}$$

5) 网络参数

$$I_{i}=B\sum_{j}a_{ij} \tag{14.4.11}$$

$$T_{ij}=-A-Ba_{ij} \tag{14.4.12}$$

从这个例子中可以看出求网络参数的又一直观方法．运动方程中的常数项就是偏流 I_{i}，一次项的系数就是 T_{ij}．

4. 图的独立集

1) 问题描述[①]

---

①详见第十一章．

图 $G(V,E)$ 的独立集是 G 的一个顶点子集 $S_1 \subseteq V$，在此子集中，任意两顶点间均无边相连．图的极大独立集问题就是要找出极大的 S_1 来．

2) **问题表示**

对于 n 个顶点的图 G 需使用 n 个神经元．网络稳定时，$V_i = 1$ 所表示的顶点 $i \in S_1$；$V_i = 0$ 则表示顶点 $i \notin S_1$．

3) **能量函数**

假定图G用邻接矩阵$A = (a_{ij})_{n \times n}$表示，则

$$E = -\frac{A}{2}\sum_i\sum_j V_i V_j + \frac{B}{2}\sum_i\sum_j a_{ij} V_i V_j \tag{14.4.13}$$

与上面的能量函数 (14.4.9) 相比，优化项差了一个符号，这是因为 (14.4.13) 所要求的是最大值．式 (14.4.9) 和 (14.4.13) 所表示的约束条件是互斥的．式 (14.4.13) 的约束条件表明：如果有边 $(i, j) \in E$，$a_{ij} = 1$，那么 V_i 或 $V_j \notin S_1$，即 $V_i = 0$ 或 $V_j = 0$ 时，约束条件最小．

4) **运动方程**

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = -u_i/\tau + A\sum_j V_j - B\sum_j a_{ij} V_j \tag{14.4.14}$$

5) **网络参数**

$$I_i = 0 \tag{14.4.15}$$

$$T_{ij} = A - Ba_{ij} \tag{14.4.16}$$

这里偏流项为 0，表示该网络不需要偏流．

5. **图的最大集团**

1) **问题描述**

图 $G(V,E)$ 的集团是 G 的一个顶点子集 $S_1 \subseteq V$，在此子集中，每两个互异的顶点间均有边关联．图的最大集团问题旨在找出使 S_1 中元素尽可能多的一个子集．

2) **问题表示**

令 G_c 是图 G 的补图（G_c 之顶点与 G 相同，G_c 之边在 G 中不出现）。如果 S_1 是 G 之集团，则 S_1 是 G_c 的独立集．假定 G 的邻接矩阵为：$A = (a_{ij})_{n \times n}$，则其补图的邻接矩阵 $A_c = (1 - a_{ij})_{n \times n}$．这样一来，为了求最大集团，就可用上面的求极大独立集的方法，只是将 a_{ij} 换成$(1 - a_{ij})$ 就行．由此可见，为了更好地把某一问题映射到神经网模型中，原问题的适当表示(如$(a_{ij})_{n \times n}, G_c$)也是非常重要的，这点在下面讨论图的平面化问题中尤其重要．

3) **能量函数**

$$E = -\frac{A}{2}\sum_i\sum_j V_i V_j + \frac{B}{2}\sum_i\sum_j \bar{a}_{ij} V_i V_j \tag{14.4.17}$$

其中$\bar{a}_{ij}=1-a_{ij}$，第一项是优化目标项，第二项为约束条件，即如果 $V_i=V_j=1$，它们都属于 S_1，那么一定有边相连使 $a_{ij}=1$，这时 $a_{ij}=0$ 取最小值。

4) 运动方程

$$\frac{\mathrm{d}u_i}{\mathrm{d}t}=-\frac{u_i}{\tau}+A\sum_i V_j-B\sum_i \bar{a}_{ij}V_j \tag{14.4.18}$$

5) 网络参数

$$I_i=0 \tag{14.4.19}$$

$$T_{ij}=A-B\bar{a}_{ij} \tag{14.4.20}$$

6. 图的最大匹配

1) 问题描述

图 $G(V,E)$ 的一个匹配是 G 的一个边子集 $M\subseteq E$，在此边集中任两条边不会共享同一个顶点。图的最大匹配问题是指找出这种最大的边集。

2) 问题表示

由于这里是以边为主考虑问题，故对于 m 条边的图需要 m 个神经元。网络稳定时，神经元 i 的输出 $V_i=1$ 表示第 i 条边是匹配边。

3) 能量函数

假定图$G(V,E)$用关联矩阵表示（n为顶点数，m为边数）。

$$(b_{ij})_{n\times m}=\begin{cases}1, & \text{顶点 } i \text{ 与边 } j \text{ 相关联}\\ 0, & \text{其它}\end{cases}$$

$$E=-A\sum_i^m V_i+\frac{B}{2}\sum_i^m\sum_j^m\sum_k^n V_iV_jb_{ki}b_{kj} \tag{14.4.21}$$

其中：第一项为优化项，与上面的优化项相比，这里采用了一次函数。虽然它们的功能相同，但一次项系数不在 T_{ij} 中出现，便于 VLSI 的实现。第二项是约束项，它表示当边 i 和边 j 有一公共交点 k 时，V_i 或 V_j 至少有一个不在匹配集中，即 $V_iV_j=0$。

4) 运动方程

$$\frac{\mathrm{d}u_i}{\mathrm{d}t}=-\frac{u_i}{\tau}+A-B\sum_j^m\sum_k^n V_jb_{ki}b_{kj} \tag{14.4.22}$$

5) 网络参数

$$I_i=A \tag{14.4.23}$$

$$T_{ij}=-B\sum_k^n b_{ki}b_{kj} \tag{14.4.24}$$

7. 图的同构

1) 问题描述

图 $G(V,E)$ 与 $G'(V',E')$ 是同构的，当且仅当存在着一一映射函数 $f:V\to V'$，使得

$\forall V_i \in V$, $V_X \in V'$, $f(V_i) = V_X$，且如果边$(V_i, V_j) \in E$，则$(V_X, V_Y) \in E$，其中$f(V_j) = V_Y$。

2) 问题表示

图的同构问题也可用$n \times n$的置换矩阵表示．网络稳定时，若神经元的输出$V_{iX} = 1$，则表示G的顶点V_i映射到G'的顶点V_X上．如果在一定的时间内网络不稳定或得到的是不满足约束条件的非法解，则说明G与G'同构的可能性不大。

3) 能量函数

$$E = \frac{A}{2}\sum_i \sum_X \sum_{Y \neq X} V_{iX} V_{iY} + \frac{B}{2}\sum_i \sum_{j \neq i} \sum_X V_{iX} V_{jX} + \frac{C}{2}\Big(\sum_i \sum_X V_{iX} - n\Big)^2$$

$$+ \frac{D}{2}\sum_X \sum_Y \sum_i \sum_j |a_{ij} - b_{XY}| V_{iX} V_{jY} \tag{14.4.25}$$

其中$(a_{ij})_{n \times n}$和$(b_{XY})_{n \times n}$分别是图G和G'的邻接矩阵。式(14.4.25)中的前三项与TSP中的约束项类似，表示一个置换矩阵．第四项是优化目标项，它表示仅当$f(V_i) = V_x$，$f(V_j) = V_y$时，存在着边$(i, j) \in E$和$(X, Y) \in E'$，使得表达式$|a_{ij} - b_{XY}| \cdot V_{iX} V_{jY}$为零取得最小值．

4) 运动方程

$$\frac{du_{Xi}}{dt} = -\frac{u_{Xi}}{\tau} - A\sum_{Y \neq X} V_{iY} - B\sum_{j \neq i} V_{jX} - C\Big(\sum_j \sum_Y V_{jY} - n\Big)$$

$$- D\sum_j \sum_Y |a_{ij} - b_{XY}| \cdot V_{jY} \tag{14.4.26}$$

5) 网络参数

$$I_{Xi} = CN \tag{14.4.27}$$

$$T_{iX,jY} = -A\delta_{ij}(1 - \delta_{XY}) - B\delta_{XY}(1 - \delta_{ij}) - C - D|a_{ij} - b_{XY}| \tag{14.4.28}$$

8. 图的着色

1) 问题描述[①]

图$G(V,E)$的顶点着色系指图中有边相连的顶点应着不同的颜色．图的着色问题旨在找出最少的颜色数目．假定顶点V_i的度为d_{Vi}，可定义图的度为$d_G = \max\{d_{V_i} | V_i \in V\}$。第12章已经证明了最多只需$\Delta = d + 1$种颜色即可对$G$进行着色。

2) 问题表示

这里只能对着色问题进行验证，即检查是否可用K种颜色对图G进行着色．同样用$n \times K$的置换矩阵表示，不过它只有行约束，即一个顶点只能着一种颜色．网络稳定时，位于(i, X)神经元的输出$V_{iX} = 1$，表示顶点i应着第X种颜色。

①详见第十二章．

3) 能量函数

假定图G用邻接矩阵$A=(a_{ij})_{n\times n}$表示，则

$$E=\frac{A}{2}\sum_{i}^{n}\sum_{X}^{K}\sum_{Y\neq X}^{K}V_{iX}V_{iY}+\frac{B}{2}\left(\sum_{i}^{n}\sum_{X}^{K}V_{iX}-n\right)^2+C\sum_{i}^{n}\sum_{j\neq i}^{n}\sum_{X}^{K}a_{ij}V_{iX}V_{jX} \tag{14.4.29}$$

其中前两项是约束条件，第三项是优化目标项，表达式 $a_{ij}V_{iX}V_{jX}$ 表示当有边 $(i,j)\in E$ 时($a_{ij}\neq 0$)，顶点 i 和顶点 j 不能着相同的颜色 X，即 V_{iX} 或 V_{jX} 为零，使得整个式子为零取最小值。

4) 运动方程

$$\frac{\mathrm{d}u_{iX}}{\mathrm{d}t}=-\frac{u_{iX}}{\tau}-A\sum_{Y\neq X}^{K}V_{iY}-B\left(\sum_{j}^{n}\sum_{Y}^{K}V_{iY}-n\right)-C\sum_{j\neq i}^{n}a_{ij}V_{jX} \tag{14.4.30}$$

5) 网络参数

$$I_{iX}=Bn \tag{14.4.31}$$

$$T_{iX,jY}=-A\delta_{ij}(1-\delta_{XY})-B-Ca_{ij}\delta_{XY}(1-\delta_{ij}) \tag{14.4.32}$$

9. 图的搜索

1) 问题描述

这里以 8 皇后问题为例说明神经网模型在图的搜索中的应用。国际象棋中皇后威力无比、控制纵、横、斜四条直线。八皇后问题就是在 8×8 的棋盘上布 8 个皇后，使得任意横线、纵线、斜对角线上只有一个皇后。

2) 问题表示

8 皇后问题也可用 8×8 的置换矩阵表示。网络稳定时，位于 (i,j) 神经元的输出 $V_{ij}=1$ 表示该位置可放一个皇后。

3) 能量函数

$$E=\frac{A}{2}\sum_{i}\sum_{X}\sum_{Y\neq X}V_{iX}V_{iY}+\frac{B}{2}\sum_{X}\sum_{i}\sum_{j\neq i}V_{iX}V_{jX}+\frac{C}{2}\sum_{i}\sum_{j\neq i}\sum_{X}\sum_{Y\neq X}\delta_{|i-j|,|X-Y|}V_{iX}V_{iY}$$
$$+\frac{D}{2}(\sum_{i}\sum_{X}V_{iX}-8)^2 \tag{14.4.33}$$

值得一提的是第三项表示两条斜对角线约束。因为当且仅当 $|i-j|=|X-Y|$ 时，(i,X) 和 (j,Y) 位于斜对角线上。这个能量函数没有优化目标项，但约束条件项还是相当复杂的。

4) 运动方程

$$\frac{\mathrm{d}u_{iX}}{\mathrm{d}t}=\frac{-u_{iX}}{\tau}-A\sum_{Y\neq X}V_{iY}-B\sum_{j\neq i}V_{jX}-C\sum_{j\neq i}\sum_{Y\neq X}\delta_{|i-j|,|X-Y|}V_{jY}-D(\sum_{j}\sum_{Y}V_{jY}-8) \tag{14.4.34}$$

5) 网络参数

$$I_{iX}=8D \tag{14.4.35}$$

$$T_{iX,jY} = -A\delta ij(1-\delta_{XY}) - B\delta_{XY}(1-\delta_{ij}) - C(1-\delta_{ij})(1-\delta_{XY})\delta_{|i-j|,|X-Y|} - D$$ (14.4.36)

10. 图的平面性测试

1) 问题描述

设有如图 14.12(a)所示的图，可重排其中各边的位置（如图 14.12(b)所示），使得

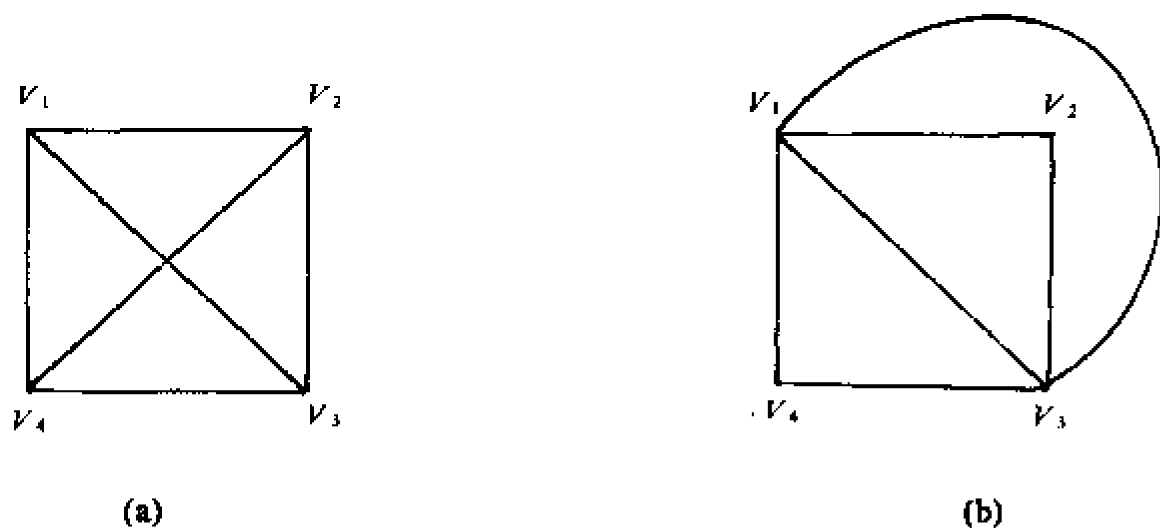


图 14.12 可平面化图

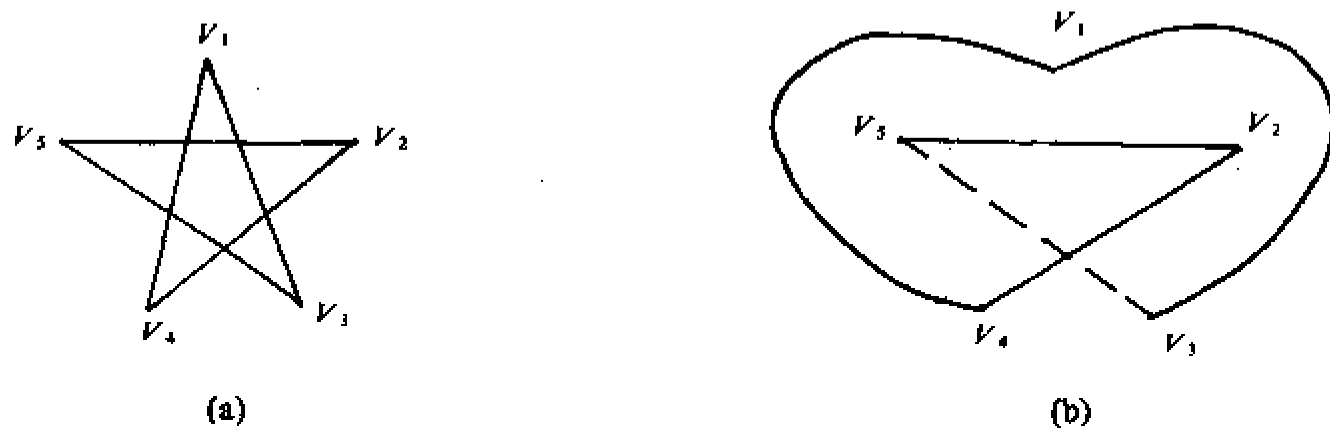


图 14.13 不可平面化图

各边不相交。但对于图 14.13(a)所示的图，无论如何改变边的位置，都不能使它们互不相交，即该图的边无法平面化。于是问题为：给定一个平面图 $G(V,E)$，能否找到一种边的连法，使它们互不相交？如果不能，最少除去几条边得到 $E'\subset E$ 使得 $G'(V,E')$能够平面化。

2) 问题表示

上面描述的问题与下述问题是等价的，即：将给定图的顶点排成一直线，如通过顶点间的上连接或下连接可将图平面化，则原图也是可以平面化的，如图 14.14 所示。因此，



图 14.14 图的直线上下连法

可用邻接矩阵形式来表示图的平面化过程[16]．若图 14.14 中有条上连边(i, j)则有矩阵的上三角元素V_{ij} $(j > i)$与之对应；同理下连边(i,j)有下三角元素V_{ij} $(i > j)$对应．图 14.14(b)的矩阵表示如下：

0	1	1	1
0	0	1	0
0	0	0	1
0	1	0	0

图 14.14(b)中有上连边 V_1V_2，则在矩阵中 $V_{12}=1$；有下连边 V_{42}，则 $V_{42}=1$．

3) **能量函数**

设图的邻接矩阵是$(a_{ij})_{n\times n}$，则

$$E=\frac{A}{2}\sum_i\sum_j(V_{ij}+V_{ij}-a_{ij})^2+\frac{B}{2}\sum_i\sum_j\sum_l\sum_m f(l,m,i,j)V_{ij}V_{lm} \tag{14.4.37}$$

这两项都是约束项．第一项表示如果顶点 i 与顶点 j 之间有连接($a_{ij}=1$)，则应有上连接或下连接，但只能是一种，否则平方项不为零．第二项表示边与边之间不应有交叉．因此，$f(l,m,i,j)$ 定义如下：

(1) 如果 $V_{ij}(i<j),V_{lm}(l<m)$ 都是上三角元素，当 $l<i<m<j$ 或 $i<l<j<m$ 时，有交叉边．这时 $f(l,m,i,j)$ 为 1 表示惩罚，如图 14.15(*a*) 和 14.15(*b*) 所示．

(2) 如果 $V_{ij}(i>j)$，$V_{lm}(l>m)$是下三角元素，当 $m<j<l<i$ 或 $j<m<i<l$ 时，有交叉边．可令 $f(l,m,i,j)=1$ 以示惩罚，如图 14.15(c)和 14.15(d)所示．

(3) 其他情况无边交叉，$f(l,m,i,j)=0$．

4) **运动方程**

$$\frac{du_{ij}}{dt}=-\frac{u_{ij}}{\tau}-A(V_{ij}+V_{ji}-a_{ij})-B\sum_l\sum_m f(l,m,i,j)\,V_{lm} \tag{14.4.38}$$

5) **网络参数**

$$I_{ij}=Aa_{ij} \tag{14.4.39}$$

$$T_{IX,JY}=-A(\delta_{ij}\delta_{XY}+\delta_{iY}\delta_{Xj})-Bf(l,m,i,j) \tag{14.4.40}$$

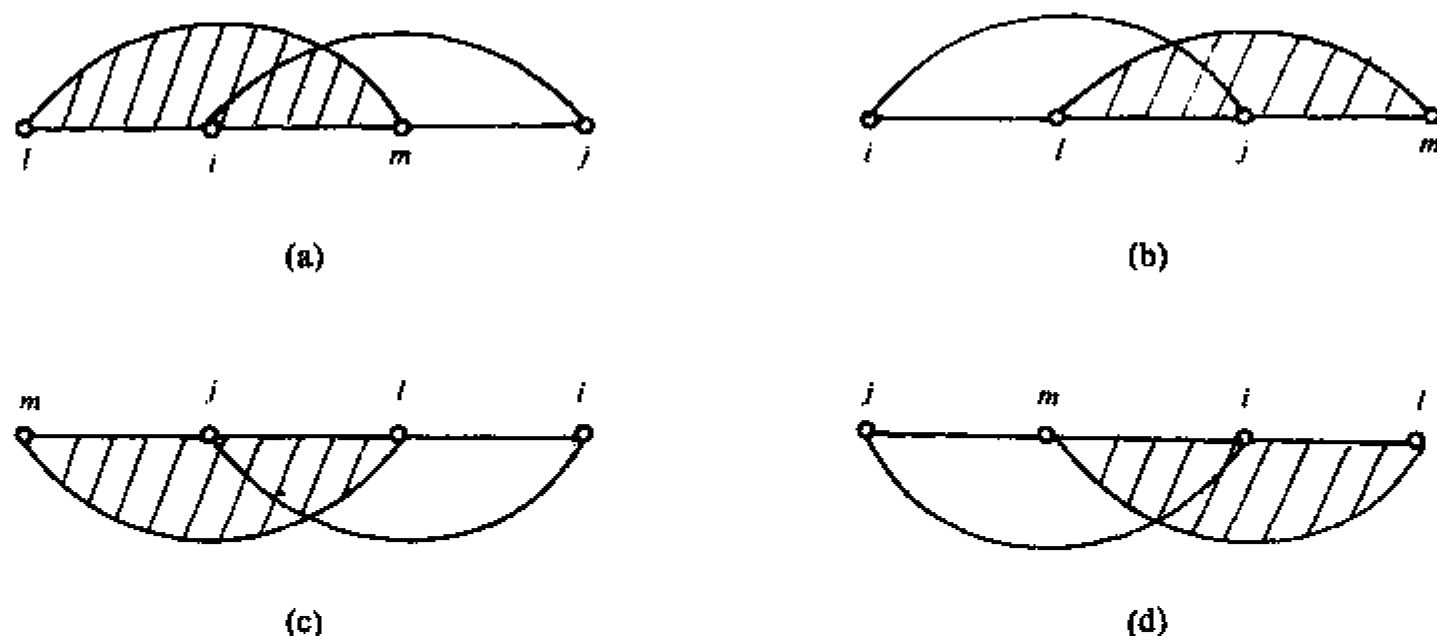


图 14.15 边交叉情况

14.5 小　　结

本章我们介绍了神经计算及其应用，着重介绍了神经网络在图论中的应用．第一节阐述了神经网络的基本原理和特征；第二节以 Hopfield 模型为例讨论了神经计算的机制；第三节以旅行商问题为线索，简单介绍了其它几种有影响的神经网络模型；最后给出了怎样用 Hopfield 模型求解 10 个典型的、难的图论问题．

参考文献

[1] Anderson J, Rosenfeld E. *Neurocomputing*, MIT Press, 1988

[2] Brandt R al. et. Alternative Networks for Solving the Traveling Salesman Problem and List Matching Problem, *Proc. IEEE Int. Conf. Neural Network* II:333–340, 1988

[3] Burr D. In Improved Elastic Net Method for the Traveling Salesman Problem, *Proc. IEEE Int. Conf. Neural Network* I: 69–76, 1988

[4] Durbin R, Willshaw D. An Analogue Approach to the Traveling Salesman Problem Using an Elastic Net Method, *Nature* 326: 689–691, 1987

[5] For J C. Solving a Combinatorial Problem via Self–Organizing Process: An Application of the Kohenen Algorithm to the Traveling Salesman Problem, Biol Cybern 59:33–40, 1988

[6] Grossberg S. *Neural Networks and Neural Intelligence*. MIT Press, 1988

[7] Hopfield J J. Neural Networks and Physical Systems with Emergent Collective Computational Abilities, *Proc. Natl. Acad. Sci.*, USA, Vol 79,2554–2558, 1982

[8] Hopfield J J. Neurons with Graded Response Have Collective Computational Properties like those of two–state neuron, *Proc. Natl. Acad. Sci.*, USA, Vol 81, 3088–3092, 1984

[9] Hopfield J J, Tank D. Neural Computations of Decisions in Optimization Problems, *Biol Cybern* 52 :141–152, 1985

[10] Kirkpatrick S, *al. et.* Optimization by Simulated Annealing, *Science* 220 :671–680, 1983

[11] Kohonen T. Self–Organization and Associate Memory, Springer Berling, 1984

[12] Lippmann R P. An Introduction to Computing with Neural Nets, *IEEE ASSP*, 4–22, April, 1987

[13] Rumelhart D E, McClelland J L. *Parallel Distributed Processing*, Vol.1–2, MIT Press, 1986

[14] Ramanujam J, Sadayappan P. Optimization by Neural Networks, *Proc. IEEE Int. Conf. Neural Network* II: 325–332, 1988

[15] Szu H. Fast TSP Algorithm Based on Binary Neuron Output and Analog Input Using the Zero–diagnoal Interconnect Matrix and Necessary and Sufficient Constraints of the Permutation Matrix, *Proc. IEEE Int. Conf. Neural Network* II: 259–266, 1988

[16] Takefuji Y, Lee K C. A Near–Optimum Parallel Planarization Algorithm, *Science*, Sep. 1989

[17] Tang X–N, et al. Equivalent Class Energy Function for Solving the TSP, *Proc. Inter. Conf. for Young Computer Scientists*, Beijing, July 1991

[18] Van den Bout DE. Improving the Performance of the Hopfield–Tank Neural Network Through Normalization and Annealing, *Biol Cybern* 62:129–139, 1989

[19] Wacholder E, et al. A Neural Network Algorithm for the Multiple Traveling Salesmen Problem, *Biol Cybern* 61: 11–19, 1989

[20] Wilson G, Pawley G. On the Stability of the Traveling Salesman Algorithm of the Hopfield and Tank, *Biol Cebern* 58: 63–70, 1988

[21] 陈国良，神经网络用于求解组合优化问题，C^2N^2–90中国神经网络首届学术大会论文集，中国·北京，122–129，1990

[22] 焦李成，神经网络系统理论，西安电子科技大学出版社，1990

算 法 索 引

责任编辑：黄　德
封面设计：王瑞荣

《[illegible]》[illegible]

· [illegible] 《[illegible]——[illegible]》[illegible]

· [illegible] 《[illegible]》[illegible]

· [illegible] 《[illegible]》[illegible]

[illegible]